Oracle8*i*

SQL Reference

Release 8.1.5

February 1999

Part No. A67779-01

ORACLE

SQL Reference, Release 8.1.5

Part No. A67779-01

Primary Authors: Diana Lorentz, Denise Oertel

Contributors: Alan Downing, Alex Tsukerman, Alok Pareek, Amit Ganesh, Andre Kruglikov, Andrew Witkowski, Angela Amor, AnhTuan Tran, Ann Rhee, Aravind Yalamanchi, Ari Mozes, Arvind Nithrakashyap, Ashok Joshi, Ashwini Surpur, Bhaskar Himatsingka, Bill Courington, Bill Waddington, Brajesh Goyal, Cetin Ozbutun, Chin-Heng Hong, Chon Lei, Daniel Wong, Dinesh Das, Edward Waugh, Eric Magrath, Franco Putzolu, Guhan Viswanathan, Harry Sun, Harvey Eneman, Jack Raitto, Jags Srinivasan, Janaki Krishnaswami, Jerry Schwarz, Jianping Yang, Jim Finnerty, John Haydu, Joyo Wijaya, Juan Tellez, Karuna Muthiah, Lilian Hobbs, Lois Price, Maria Pratt, Mark Jungerman, Michael Depledge, Mohamed Zait, Muralidhar Krishnaprasad, Namit Jain, Nipun Agarwal, Paul Justus, Paul Raveling, Qin Yu, Radhakrishna Hari, Ravi Murthy, Rick Anderson, Rick Wessman, Robert Jenkins, Rosanne Park, Sanjay Kaluskar, Sankar Subramanian, Sophia Yeung, Sriram Samu, Steve Vivian, Subramanian Muralidhar, Sukhjit Singh, Susan Kotsovolos, Thong Bui, Thuvan Hoang, Vikas Arora, Vinay Srihari, Vishu Krishnamurthy, Vishy Karra, Wei Huang, Wei Wang

Contents

Send Us Your Comments

SQL Reference, Release 8.1.5

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). Please send your comments to:

Server Technologies Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
Fax: (650) 506-7228

or e-mail comments to the Information Development department at the following e-mail address:

infodev@us.oracle.com

Preface

Well begun is half done.

Aristotle, *Nicomachean Ethics*

This reference contains a complete description of the Structured Query Language (SQL) used to manage information in an Oracle database. Oracle SQL is a superset of the American National Standards Institute (ANSI) and the International Standards Organization (ISO) SQL92 standard at entry level conformance.

For information on PL/SQL, Oracle's procedural language extension to SQL, see *PL/SQL User's Guide and Reference.*

Detailed descriptions of Oracle embedded SQL can be found in the *Pro*C/C++ Precompiler Programmer's Guide, SQL*Module for Ada Programmer's Guide*, and the *Pro*COBOL Precompiler Programmer's Guide*.

Features and Functionality

Oracle8i SQL Reference contains information about the features and functionality of the Oracle8*i* and the Oracle8*i* Enterprise Edition products. Oracle8*i* and Oracle8*i* Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional.

For information about the differences between Oracle8*i* and the Oracle8*i* Enterprise Edition and the available features and options, see *Getting to Know Oracle8i*. That book also describes all the features that are new in Oracle8*i*.

Audience

This reference is intended for all users of Oracle SQL.

How this Reference Is Organized

This reference is divided into the following parts:

Volume 1

Chapter 1, "Introduction"

This chapter defines SQL and describes its history as well as the advantages of using it to access relational databases.

Chapter 2, "Basic Elements of Oracle SQL"

This chapter describes the basic building blocks of an Oracle database and Oracle SQL.

Chapter 3, "Operators"

This chapter describes how to use SQL operators to combine data into expressions and conditions.

Chapter 4, "Functions"

This chapter describes how to use SQL functions to combine data into expressions and conditions.

Chapter 5, "Expressions, Conditions, and Queries"

This chapter describes SQL expressions and conditions and discusses the various ways of extracting information from your database through queries.

Chapter 6, "About SQL Statements"

This chapter lists the various types of SQL statements, and provides a table to help you find the appropriate SQL statement for your database task.

Volume 2

Chapter 7, "SQL Statements"

This chapter lists and describes all Oracle SQL statements in alphabetical order.

Appendix A, "Syntax Diagrams"

This appendix describes how to read the syntax diagrams in this reference.

Appendix B, "Oracle and Standard SQL"

This appendix describes Oracle compliance with ANSI and ISO standards.

Appendix C, "Oracle Reserved Words"

This appendix lists words that are reserved for internal use by Oracle.

What's New in This Release?

Oracle8*i* contains many new features, which are documented throughout this reference. For a description of all features new to this release, see *Getting to Know Oracle8i*.

The following top-level SQL statements are new to Oracle8*i*:

- ALTER DIMENSION on page 7-24
- ALTER JAVA on page 7-43
- ALTER OUTLINE on page 7-58
- ASSOCIATE STATISTICS on page 7-194
- CALL on page 7-210
- CREATE CONTEXT on page 7-243
- CREATE DIMENSION on page 7-259
- CREATE INDEXTYPE on page 7-291
- CREATE JAVA on page 7-293
- CREATE OPERATOR on page 7-320
- CREATE OUTLINE on page 7-323
- CREATE TEMPORARY TABLESPACE on page 7-399
- DISASSOCIATE STATISTICS on page 7-444
- DROP CONTEXT on page 7-448
- DROP DIMENSION on page 7-450
- DROP INDEXTYPE on page 7-456

- DROP JAVA on page 7-457
- DROP OPERATOR on page 7-463
- DROP OUTLINE on page 7-464

In addition, users familiar with the Release 8.0 documentation will find that the following sections have been moved or renamed:

- The section "Format Models" now appears in Chapter 2 on page 2-33.
- Chapter 3 has been divided into several smaller chapters:
  - Chapter 3, "Operators"
  - Chapter 4, "Functions"
  - Chapter 5, "Expressions, Conditions, and Queries". The last section, "Queries and Subqueries" on page 5-18, provides background for the syntactic and semantic information in "SELECT and Subqueries" on page 7-541.
- A new chapter, Chapter 6, "About SQL Statements", has been added to help you find the correct SQL statement for a particular task.
- The *archive_log_clause* is no longer a separate section, but has been incorporated into "ALTER SYSTEM" on page 7-95.
- The *deallocate_unused_clause* is no longer a separate section, but has been incorporated into "ALTER TABLE" on page 7-113, "ALTER CLUSTER" on page 7-2, and "ALTER INDEX" on page 7-29.
- The *disable_clause* is no longer a separate section, but has been incorporated into "CREATE TABLE" on page 7-359 and "ALTER TABLE" on page 7-113.
- The *drop_clause* is no longer a separate section. It has become the *drop_constraint_clause* of the ALTER TABLE statement (to distinguish it from the new *drop_column_clause* of that statement). See "ALTER TABLE" on page 7-113.
- The *enable_clause* is no longer a separate section, but has been incorporated into "CREATE TABLE" on page 7-359 and "ALTER TABLE" on page 7-113.
- The *parallel_clause* is no longer a separate section. The clause has been simplified, and has been incorporated into the various statements where it is relevant.
- The *recover_clause* is no longer a separate section. Recovery functionality has been enhanced, and because it is always implemented through the ALTER

DATABASE statement, it has been incorporated into that section. See "ALTER DATABASE" on page 7-6.

- The sections on **snapshots** and **snapshot logs** have been moved and renamed. Snapshot functionality has been greatly enhanced, and these objects are now called **materialized views**. See "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 300, "ALTER MATERIALIZED VIEW / SNAPSHOT" on page 7-45, "DROP MATERIALIZED VIEW / SNAPSHOT" on page 7-459, "CREATE MATERIALIZED VIEW LOG / SNAPSHOT LOG" on page 7-314, "ALTER MATERIALIZED VIEW LOG / SNAPSHOT LOG" on page 7-54, and "DROP MATERIALIZED VIEW LOG / SNAPSHOT LOG" on page 7-461.
- The section on **subqueries** has now been combined with the SELECT statement. See "SELECT and Subqueries" on page 7-541.

Conventions Used in this Reference

This section explains the conventions used in this book including:

- Text
- Syntax Diagrams and Notation
- Code Examples
- Example Data

Text

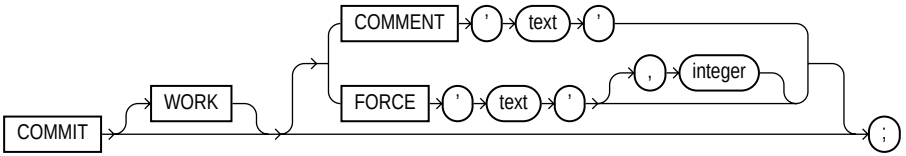
The text in this reference adheres to the following conventions:

UPPERCASE	Uppercase text calls attention to SQL keywords, filenames, and initialization parameters.
italics	Italicized text calls attention to parameters of SQL statements.
boldface	Boldface text calls attention to definitions of terms.

Syntax Diagrams and Notation

Syntax Diagrams This reference uses syntax diagrams to show SQL statements in Chapter 7, “SQL Statements”, and to show other elements of the SQL language in Chapter 2, “Basic Elements of Oracle SQL”, Chapter 3, “Operators”, Chapter 4,

"Functions", and Chapter 5, "Expressions, Conditions, and Queries". These syntax diagrams use lines and arrows to show syntactic structure, as shown here:



If you are not familiar with this type of syntax diagram, refer to Appendix A, "Syntax Diagrams", for a description of how to read them. This section describes the components of syntax diagrams and gives examples of how to write SQL statements. Syntax diagrams are made up of these items:

Keywords Keywords have special meanings in the SQL language. In the syntax diagrams, keywords appear in UPPERCASE. You must use keywords in your SQL statements exactly as they appear in the syntax diagram, except that they can be either uppercase or lowercase. For example, you must use the CREATE keyword to begin your CREATE TABLE statements just as it appears in the CREATE TABLE syntax diagram.

Parameters Parameters act as placeholders in syntax diagrams. They appear in lowercase. Parameters are usually names of database objects, Oracle datatype names, or expressions. When you see a parameter in a syntax diagram, substitute an object or expression of the appropriate type in your SQL statement. For example, to write a CREATE TABLE statement, use the name of the table you want to create, such as EMP, in place of the *table* parameter in the syntax diagram. (Note that parameter names appear in italics in the text.)

This lists shows parameters that appear in the syntax diagrams and provides examples of the values you might substitute for them in your statements:

Parameter	Description	Examples
table	The substitution value must be the name of an object of the type specified by the parameter. For a list of all types of objects, see the section, "Schema Objects" on page 2-63.	`emp`
c	The substitution value must be a single character from your database character set.	`T` `s`

Parameter	Description	Examples
'text'	The substitution value must be a text string in single quotes. See the syntax description of '*text*' in "Text" on page 2-2.	`'Employee records'`
char	The substitution value must be an expression of datatype CHAR or VARCHAR2 or a character literal in single quotes.	`ename` `'Smith'`
condition	The substitution value must be a condition that evaluates to TRUE or FALSE. See the syntax description of *condition* in "Conditions" on page 5-13.	`ename >'A'`

date
d
	The substitution value must be a date constant or an expression of DATE datatype.	`TO_DATE(`
		`'01-Jan-1994',`
		`'DD-MON-YYYY')`

expr	The substitution value can be an expression of any datatype as defined in the syntax description of *expr* in "Expressions" on page 5-1.	`sal + 1000`
integer	The substitution value must be an integer as defined by the syntax description of integer in "Integer" on page 2-3.	`72`
number *m* *n*	The substitution value must be an expression of NUMBER datatype or a number constant as defined in the syntax description of *number* in "Number" on page 2-4.	`AVG(sal)` `15 * 7`
raw	The substitution value must be an expression of datatype RAW.	`HEXTORAW('7D')`
subquery	The substitution value must be a SELECT statement that will be used in another SQL statement. See "SELECT and Subqueries" on page 7-541.	`SELECT ename` `FROM emp`
db_name	The substitution value must be the name of a nondefault database in an embedded SQL program.	`sales_db`

Parameter	Description	Examples
db_string	The substitution value must be the database identification string for a Net8 database connection. For details, see the user's guide for your specific Net8 protocol.	

Code Examples

This reference contains many examples of SQL statements. These examples show you how to use elements of SQL. The following example shows a CREATE TABLE statement:

```
CREATE TABLE accounts
    ( accno     NUMBER,
      owner     VARCHAR2(10),
      balance   NUMBER(7,2) );
```

Note that examples appear in a different font than the text.

Examples follow these conventions:

- Keywords, such as CREATE and NUMBER, appear in uppercase.
- Names of database objects and their parts, such as ACCOUNTS and ACCNO, appear in lowercase, although they appear in uppercase in the text.
- PL/SQL blocks appear in italics. Keywords and parameters in these blocks may not be documented in this reference unless they are also SQL keywords and parameters. For more information see *PL/SQL User's Guide and Reference.*

SQL is not case sensitive (except for quoted identifiers), so you need not follow these conventions when writing your own SQL statements. However, your statements may be easier for you to read if you do.

Some Oracle tools require you to terminate SQL statements with a special character. For example, the code examples in this reference were issued through SQL*Plus, and therefore are terminated with a semicolon (;). If you issue these example statements to Oracle, you must terminate them with the special character expected by the Oracle tool you are using.

Example Data

Many of the examples in this reference operate on sample tables. The definitions of some of these tables appear in a SQL script available on your distribution medium.

On most operating systems the name of this script is UTLSAMPL.SQL, although its exact name and location depend on your operating system. This script creates sample users and creates these sample tables in the schema of the user SCOTT:

```
CREATE TABLE dept
    (deptno    NUMBER(2)      CONSTRAINT pk_dept PRIMARY KEY,
     dname     VARCHAR2(14),
     loc       VARCHAR2(13) );
CREATE TABLE emp
    (empno     NUMBER(4)      CONSTRAINT pk_emp PRIMARY KEY,
     ename     VARCHAR2(10),
     job       VARCHAR2(9),
     mgr       NUMBER(4),
     hiredate  DATE,
     sal       NUMBER(7,2),
     comm      NUMBER(7,2),
     deptno    NUMBER(2)    CONSTRAINT fk_deptno REFERENCES dept );
CREATE TABLE bonus
    (ename     VARCHAR2(10),
     job       VARCHAR2(9),
     sal       NUMBER,
     comm      NUMBER );
CREATE TABLE salgrade
    (grade     NUMBER,
     losal     NUMBER,
     hisal     NUMBER );
```

The script also fills the sample tables with this data:

```
SELECT * FROM dept;

DEPTNO  DNAME      LOC
------- ---------- ---------
10      ACCOUNTING NEW YORK
20      RESEARCH   DALLAS
30      SALES      CHICAGO
40      OPERATIONS BOSTON

SELECT * FROM emp;

EMPNO  ENAME   JOB          MGR  HIREDATE    SAL   COMM  DEPTNO
-----  ------- --------- ------ --------- ------ ------ -------
 7369  SMITH   CLERK       7902 17-DEC-80    800             20
 7499  ALLEN   SALESMAN    7698 20-FEB-81   1600    300      30
 7521  WARD    SALESMAN    7698 22-FEB-81   1250    500      30
 7566  JONES   MANAGER     7839 02-APR-81   2975             20
```

```
 7654  MARTIN  SALESMAN    7698 28-SEP-81   1250   1400      30
 7698  BLAKE   MANAGER     7839 01-MAY-81   2850              30
 7782  CLARK   MANAGER     7839 09-JUN-81   2450              10
 7788  SCOTT   ANALYST     7566 19-APR-87   3000              20
 7839  KING    PRESIDENT         17-NOV-81   5000              10
 7844  TURNER  SALESMAN    7698 08-SEP-81   1500              30
 7876  ADAMS   CLERK       7788 23-MAY-87   1100              20
 7900  JAMES   CLERK       7698 03-DEC-81    950              30
 7902  FORD    ANALYST     7566 03-DEC-81   3000              20
 7934  MILLER  CLERK       7782 23-JAN-82   1300              10

SELECT * FROM salgrade;

GRADE LOSAL HISAL
----- ----- -----
1     700   1200
2     1201  1400
3     1401  2000
4     2001  3000
5     3001  9999
```

To perform all the operations of the script, run it when you are logged into Oracle as the user SYSTEM.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of our references. As we write, revise, and evaluate, your opinions are the most important input we receive. At the front of this reference is a reader's comment form that we encourage you to use to tell us both what you like and what you dislike about this (or other) Oracle manuals. If the form is missing, or you would like to contact us, please use the following address or fax number:

Server Technologies Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065
FAX: 650-506-7228

You can also e-mail your comments to the Information Development department at the following e-mail address: infodev@us.oracle.com

1

Introduction

The chief merit of language is clearness . . . .

Galen, *On the Natural Faculties*

Structured Query Language (SQL) is the set of statements with which all programs and users access data in an Oracle database. Application programs and Oracle tools often allow users access to the database without using SQL directly, but these applications in turn must use SQL when executing the user's request. This chapter provides background information on SQL as used by most relational database systems. Topics include:

- History of SQL
- SQL Standards
- Embedded SQL
- Lexical Conventions
- Tools Support

History of SQL

Dr. E. F. Codd published the paper, "A Relational Model of Data for Large Shared Data Banks", in June 1970 in the Association of Computer Machinery (ACM) journal, *Communications of the ACM*. Codd's model is now accepted as the definitive model for relational database management systems (RDBMS). The language, Structured English Query Language ("SEQUEL") was developed by IBM Corporation, Inc., to use Codd's model. SEQUEL later became SQL (still pronounced "sequel"). In 1979, Relational Software, Inc. (now Oracle Corporation) introduced the first commercially available implementation of SQL. Today, SQL is accepted as the standard RDBMS language.

SQL Standards

Oracle SQL complies with industry-accepted standards. Oracle Corporation ensures future compliance with evolving SQL standards by participating actively in SQL standards committees. Industry-accepted committees are the American National Standards Institute (ANSI) and the International Standards Organization (ISO), which is affiliated with the International Electrotechnical Commission (IEC). Both ANSI and the ISO / IEC have accepted SQL as the standard language for relational databases. When a new SQL standard is simultaneously published by these organizations, the names of the standards conform to conventions used by the organization, but the standards are technically identical.

The latest SQL standard published by ANSI and ISO is often called SQL92 (and sometimes SQL2). The formal names of the new standard are:

- ANSI X3.135-1992, "Database Language SQL"
- ISO / IEC 9075:1992, "Database Language SQL"

SQL92 defines four levels of compliance: Entry, Transitional, Intermediate, and Full. A conforming SQL implementation must support at least Entry SQL. Oracle8*i* fully supports Entry SQL and has many features that conform to Transitional, Intermediate, or Full SQL.

Oracle8*i* is 100% compliant with Entry-level SQL92 as outlined in Federal Information Processing Standard (FIPS) PUB 127-2.

> **Additional Information:** For more information about Oracle and standard SQL, see Appendix B, "Oracle and Standard SQL".

How SQL Works

The strengths of SQL provide benefits for all types of users, including application programmers, database administrators, managers, and end users. Technically speaking, SQL is a data sublanguage. The purpose of SQL is to provide an interface to a relational database such as Oracle, and all SQL statements are instructions to the database. In this SQL differs from general-purpose programming languages like C and BASIC. Among the features of SQL are the following:

- It processes sets of data as groups rather than as individual units.
- It provides automatic navigation to the data.
- It uses statements that are complex and powerful individually, and that therefore stand alone. Flow-control statements were not part of SQL originally, but they are found in the recently accepted optional part of SQL, ISO / IEC

9075-5: 1996. Flow-control statements are commonly known as "persistent stored modules" (PSM), and Oracle's PL/SQL extension to SQL is similar to PSM.

Essentially, SQL lets you work with data at the logical level. You need to be concerned with the implementation details only when you want to manipulate the data. For example, to retrieve a set of rows from a table, you define a condition used to filter the rows. All rows satisfying the condition are retrieved in a single step and can be passed as a unit to the user, to another SQL statement, or to an application. You need not deal with the rows one by one, nor do you have to worry about how they are physically stored or retrieved. All SQL statements use the **optimizer**, a part of Oracle that determines the most efficient means of accessing the specified data. Oracle also provides techniques you can use to make the optimizer perform its job better.

SQL provides statements for a variety of tasks, including:

- Querying data
- Inserting, updating, and deleting rows in a table
- Creating, replacing, altering, and dropping objects
- Controlling access to the database and its objects
- Guaranteeing database consistency and integrity

SQL unifies all of the above tasks in one consistent language.

Common Language for All Relational Databases

All major relational database management systems support SQL, so you can transfer all skills you have gained with SQL from one database to another. In addition, all programs written in SQL are portable. They can often be moved from one database to another with very little modification.

Embedded SQL

Embedded SQL refers to the use of standard SQL statements embedded within a procedural programming language. The embedded SQL statements are documented in the Oracle precompiler books, *SQL*Module for Ada Programmer's Guide*, *Pro*C/C++ Precompiler Programmer's Guide*, and *Pro*COBOL Precompiler Programmer's Guide*.

Embedded SQL is a collection of these statements:

- All SQL commands, such as SELECT and INSERT, available with SQL with interactive tools
- Dynamic SQL execution commands, such as PREPARE and OPEN, which integrate the standard SQL statements with a procedural programming language

Embedded SQL also includes extensions to some standard SQL statements. Embedded SQL is supported by the Oracle precompilers. The Oracle precompilers interpret embedded SQL statements and translate them into statements that can be understood by procedural language compilers.

Each of these Oracle precompilers translates embedded SQL programs into a different procedural language:

- Pro*C/C++ precompiler
- Pro*COBOL precompiler
- SQL*Module for ADA

> **Additional Information:** For a definition of the Oracle precompilers and the embedded SQL statements, see *SQL*Module for Ada Programmer's Guide*, *Pro*C/C++ Precompiler Programmer's Guide*, and *Pro*COBOL Precompiler Programmer's Guide*.

Lexical Conventions

The following lexical conventions for issuing SQL statements apply specifically to Oracle's implementation of SQL, but are generally acceptable in other SQL implementations.

When you issue a SQL statement, you can include one or more tabs, carriage returns, spaces, or comments anywhere a space occurs within the definition of the statement. Thus, Oracle evaluates the following two statements in the same manner:

```
SELECT ENAME,SAL*12,MONTHS_BETWEEN(HIREDATE,SYSDATE) FROM EMP;

SELECT ENAME,
     SAL * 12,
          MONTHS_BETWEEN( HIREDATE, SYSDATE )
FROM EMP;
```

Case is insignificant in reserved words, keywords, identifiers and parameters. However, case is significant in text literals and quoted names. See the syntax description in "Text" on page 2-2.

Tools Support

Most (but not all) Oracle tools support all features of Oracle's SQL. This reference describes the complete functionality of SQL. If the Oracle tool that you are using does not support this complete functionality, you can find a discussion of the restrictions in the manual describing the tool, such as *SQL*Plus User's Guide and Reference*.

If you are using Trusted Oracle, see your Trusted Oracle documentation for information about SQL statements specific to that environment.

2

Basic Elements of Oracle SQL

Once the whole is divided, the parts need names.

Lao Tsu, *Tao Te Ching: Thirty-Two*

This chapter contains reference information on the basic elements of Oracle SQL. These elements are simplest building blocks of SQL statements. Therefore, before using the statements described in Chapter 7, "SQL Statements", you should familiarize yourself with the concepts covered in this chapter, as well as in Chapter 3, "Operators", Chapter 4, "Functions", Chapter 5, "Expressions, Conditions, and Queries", and Chapter 6, "About SQL Statements":

- Literals
  - Text
  - Integer
  - Number
- Datatypes
- Format Models
- Nulls
- Pseudocolumns
- Comments
- Database Objects
- Schema Object Names and Qualifiers
- Referring to Schema Objects and Parts

Literals

The terms **literal** and **constant value** are synonymous and refer to a fixed data value. For example, 'JACK', 'BLUE ISLAND', and '101' are all character literals; 5001 is a numeric literal. Note that character literals are enclosed in single quotation marks, which enable Oracle to distinguish them from schema object names.

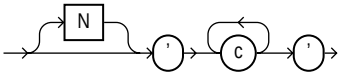
Many SQL statements and functions require you to specify character and numeric literal values. You can also specify literals as part of expressions and conditions. You can specify character literals with the '*text*' notation, national character literals with the *N'text'* notation, and numeric literals with the *integer* or *number* notation, depending on the context of the literal. The syntactic forms of these notations appear in the sections that follow.

Text

Text specifies a text or character literal. You must use this notation to specify values whenever '*text*' or *char* appear in expressions, conditions, SQL functions, and SQL statements in other parts of this reference.

The syntax of text is as follows:

text::=



where

N — specifies representation of the literal using the national character set. Text entered using this notation is translated into the national character set by Oracle when used.

c — is any member of the user's character set, except a single quotation mark (').

' ' — are two single quotation marks that begin and end text literals. To represent one single quotation mark within a literal, enter two single quotation marks.

A text literal must be enclosed in single quotation marks. This reference uses the terms **text literal** and **character literal** interchangeably.

Text literals have properties of both the CHAR and VARCHAR2 datatypes:

- Within expressions and conditions, Oracle treats text literals as though they have the datatype CHAR by comparing them using blank-padded comparison semantics. See "Blank-Padded Comparison Semantics" on page 2-28.
- A text literal can have a maximum length of 4000 bytes.

Here are some valid text literals:

```
'Hello'
'ORACLE.dbs'
'Jackie''s raincoat'
'09-MAR-98'
N'nchar literal'
```

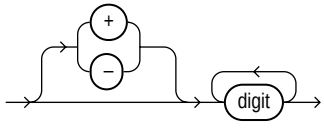
For more information, see the syntax description of *expr* in "Expressions" on page 5-1.

Integer

You must use the integer notation to specify an integer whenever *integer* appears in expressions, conditions, SQL functions, and SQL statements described in other parts of this reference.

The syntax of *integer* is as follows:

integer::=



where

digit is one of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

An integer can store a maximum of 38 digits of precision.

Here are some valid integers:

```
7
+255
```

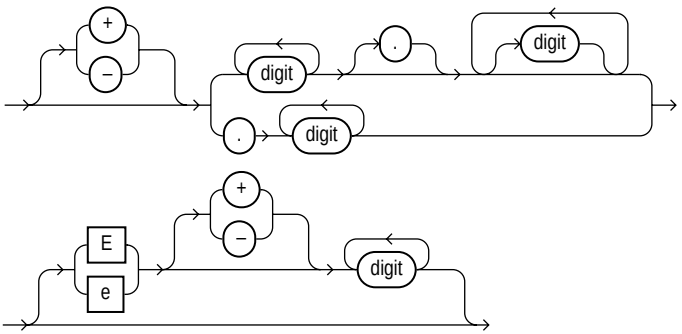
For more information, see the syntax description of *expr* in "Expressions" on page 5-1.

Number

You must use the number notation to specify values whenever *number* appears in expressions, conditions, SQL functions, and SQL statements in other parts of this reference.

The syntax of *number* is as follows:

number::=



where

+, - indicates a positive or negative value. If you omit the sign, a positive value is the default.

digit is one of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9.

e, E indicates that the number is specified in scientific notation. The digits after the E specify the exponent. The exponent can range from -130 to 125.

A *number* can store a maximum of 38 digits of precision.

If you have established a decimal character other than a period (.) with the initialization parameter NLS_NUMERIC_CHARACTERS, you must specify numeric literals with '*text*' notation. In such cases, Oracle automatically converts the text literal to a numeric value.

For example, if the NLS_NUMERIC_CHARACTERS parameter specifies a decimal character of comma, specify the number 5.123 as follows:

```
'5,123'
```

For more information on this parameter, see "ALTER SESSION" on page 7-78 and *Oracle8i Reference*.

Here are some valid representations of *number*:

```
25
+6.34
0.5
25e-03
-1
```

For more information, see the syntax description of *expr* in "Expressions" on page 5-1.

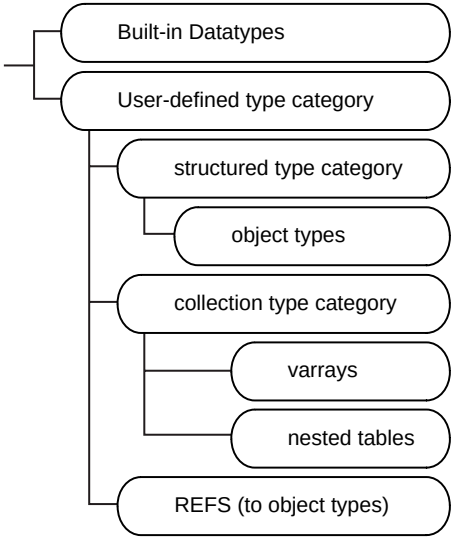
Datatypes

Each literal or column value manipulated by Oracle has a **datatype**. A value's datatype associates a fixed set of properties with the value. These properties cause Oracle to treat values of one datatype differently from values of another. For example, you can add values of NUMBER datatype, but not values of RAW datatype.

When you create a table or cluster, you must specify a datatype for each of its columns. When you create a procedure or stored function, you must specify a datatype for each of its arguments. These datatypes define the domain of values that each column can contain or each argument can have. For example, DATE columns cannot accept the value February 29 (except for a leap year) or the values 2 or 'SHOE'. Each value subsequently placed in a column assumes the column's datatype. For example, if you insert '01-JAN-98' into a DATE column, Oracle treats the '01-JAN-98' character string as a DATE value after verifying that it translates to a valid date.

Oracle provides a number of built-in datatypes as well as several categories for user-defined types, as shown in Figure 2–1.

Figure 2–1 Oracle Type Categories

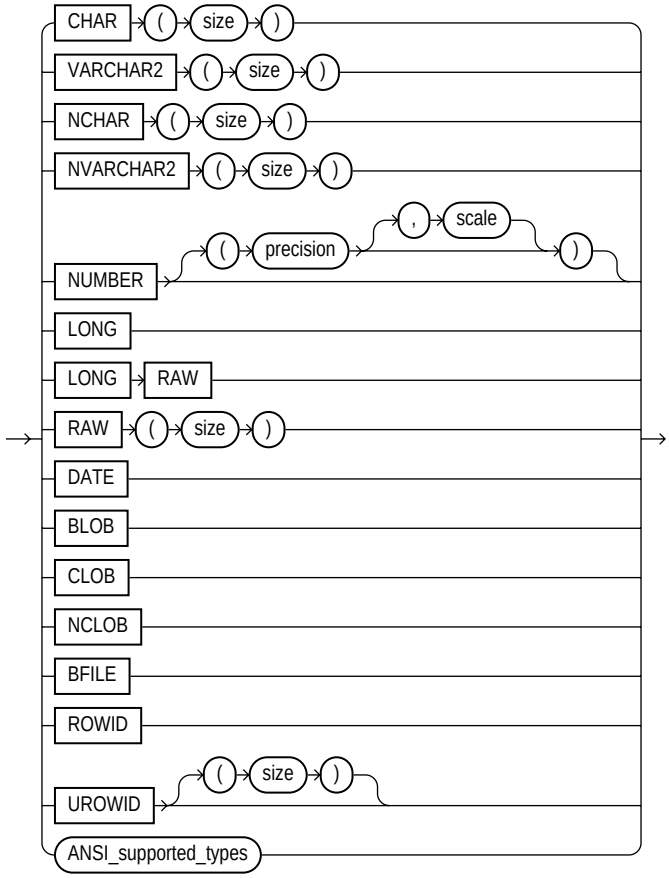


The syntax of the Oracle built-in datatypes appears in the next diagram. Table 2–1 summarizes Oracle built-in datatypes. The rest of this section describes these datatypes as well as the various kinds of user-defined types.

Note: The Oracle precompilers recognize other datatypes in embedded SQL programs. These datatypes are called *external datatypes* and are associated with host variables. Do not confuse built-in and user-defined datatypes with external datatypes. For information on external datatypes, including how Oracle converts between them and built-in or user-defined datatypes, see *Pro*COBOL Precompiler Programmer's Guide*, *Pro*C/C++ Precompiler Programmer's Guide*, and *SQL*Module for Ada Programmer's Guide*.

---

built-in datatypes:



The ANSI-supported datatypes appear in the figure that follows. Table 2–2 shows the mapping of ANSI-supported datatypes to Oracle build-in datatypes.

ANSI-supported datatypes:

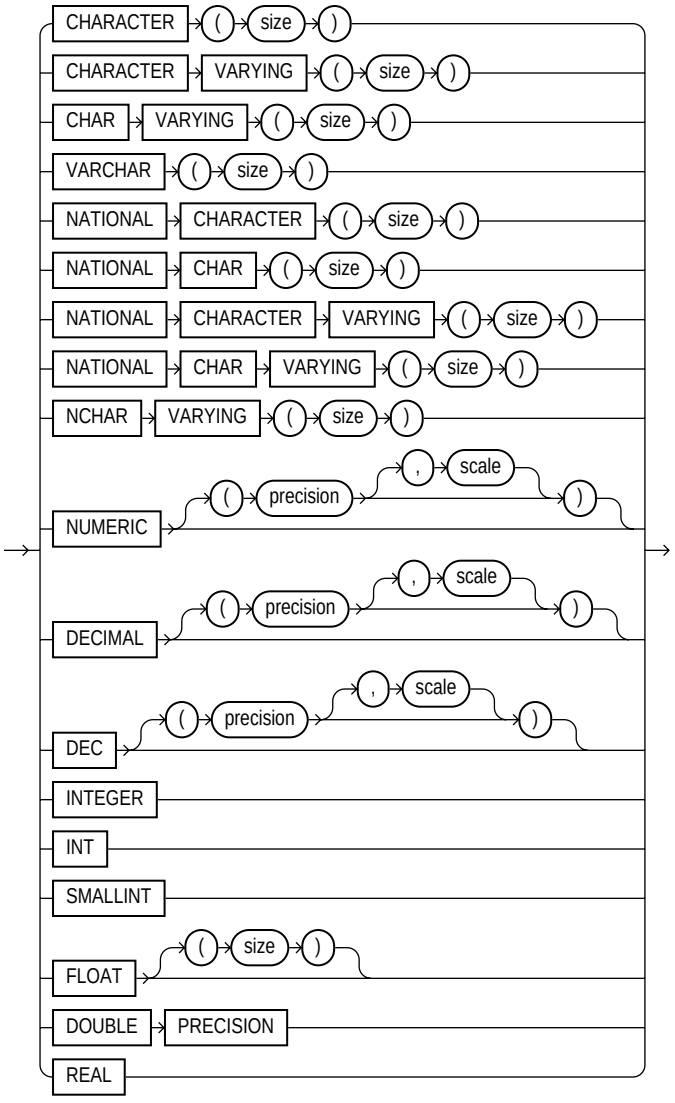


Table 2–1 Built-In Datatype Summary

Code[a]	Built-In Datatype	Description
1	VARCHAR2(*size*)	Variable-length character string having maximum length *size* bytes. Maximum *size* is 4000, and minimum is 1. You must specify *size* for VARCHAR2.
1	NVARCHAR2(*size*)	Variable-length character string having maximum length *size* characters or bytes, depending on the choice of national character set. Maximum *size* is determined by the number of bytes required to store each character, with an upper limit of 4000 bytes. You must specify *size* for NVARCHAR2.
2	NUMBER(*p*,*s*)	Number having precision *p* and scale *s*. The precision *p* can range from 1 to 38. The scale *s* can range from -84 to 127.
8	LONG	Character data of variable length up to 2 gigabytes, or 2^{31} -1 bytes.
12	DATE	Valid date range from January 1, 4712 BC to December 31, 9999 AD.
23	RAW(*size*)	Raw binary data of length *size* bytes. Maximum *size* is 2000 bytes. You must specify *size* for a RAW value.
24	LONG RAW	Raw binary data of variable length up to 2 gigabytes.
69	ROWID	Hexadecimal string representing the unique address of a row in its table. This datatype is primarily for values returned by the ROWID pseudocolumn.
208	UROWID [(size)]	Hexadecimal string representing the logical address of a row of an index-organized table. The optional *size* is the size of a column of type UROWID. The maximum size and default is 4000 bytes.
96	CHAR(*size*)	Fixed-length character data of length *size* bytes. Maximum *size* is 2000 bytes. Default and minimum *size* is 1 byte.

[a] The codes listed for the datatypes are used internally by Oracle. The datatype code of a column or object attribute is returned when you use the DUMP function.

Table 2–1 (Cont.) Built-In Datatype Summary

Code[a]	Built-In Datatype	Description
96	NCHAR(*size*)	Fixed-length character data of length *size* characters or bytes, depending on the choice of national character set. Maximum *size* is determined by the number of bytes required to store each character, with an upper limit of 2000 bytes. Default and minimum *size* is 1 character or 1 byte, depending on the character set.
112	CLOB	A character large object containing single-byte characters. Both fixed-width and variable-width character sets are supported, both using the CHAR database character set. Maximum size is 4 gigabytes.
112	NCLOB	A character large object containing multibyte characters. Both fixed-width and variable-width character sets are supported, both using the NCHAR database character set. Maximum size is 4 gigabytes. Stores national character set data.
113	BLOB	A binary large object. Maximum size is 4 gigabytes.
114	BFILE	Contains a locator to a large binary file stored outside the database. Enables byte stream I/O access to external LOBs residing on the database server. Maximum size is 4 gigabytes.

[a] The codes listed for the datatypes are used internally by Oracle. The datatype code of a column or object attribute is returned when you use the DUMP function.

Character Datatypes

Character datatypes store character (alphanumeric) data, which are words and free-form text, in the database character set or national character set. They are less restrictive than other datatypes and consequently have fewer properties. For example, character columns can store all alphanumeric values, but NUMBER columns can store only numeric values.

Character data is stored in strings with byte values corresponding to one of the character sets, such as 7-bit ASCII or EBCDIC Code Page 500, specified when the database was created. Oracle supports both single-byte and multibyte character sets.

These datatypes are used for character data:

- CHAR Datatype

- NCHAR Datatype
- NVARCHAR2 Datatype
- VARCHAR2 Datatype

CHAR Datatype

The CHAR datatype specifies a fixed-length character string. When you create a table with a CHAR column, you supply the column length in bytes. Oracle subsequently ensures that all values stored in that column have this length. If you insert a value that is shorter than the column length, Oracle blank-pads the value to column length. If you try to insert a value that is too long for the column, Oracle returns an error.

The default length for a CHAR column is 1 character and the maximum allowed is 2000 characters. A zero-length string can be inserted into a CHAR column, but the column is blank-padded to 1 character when used in comparisons. For information on comparison semantics, see "Datatype Comparison Rules" on page 2-27.

NCHAR Datatype

The NCHAR datatype specifies a fixed-length national character set character string. When you create a table with an NCHAR column, you define the column length either in characters or in bytes. You define the national character set when you create your database.

If the national character set of the database is fixed width, such as JA16EUCFIXED, then you declare the NCHAR column size as the number of characters desired for the string length. If the national character set is variable width, such as JA16SJIS, you declare the column size in bytes. The following statement creates a table with one NCHAR column that can store strings up to 30 characters in length using JA16EUCFIXED as the national character set:

```
CREATE TABLE tab1 (col1 NCHAR(30));
```

The column's maximum length is determined by the national character set definition. Width specifications of character datatype NCHAR refer to the number of characters if the national character set is fixed width and refer to the number of bytes if the national character set is variable width. The maximum column size allowed is 2000 bytes. For fixed-width, multibyte character sets, the maximum length of a column allowed is the number of characters that fit into no more than 2000 bytes.

If you insert a value that is shorter than the column length, Oracle blank-pads the value to column length. You cannot insert a CHAR value into an NCHAR column, nor can you insert an NCHAR value into a CHAR column.

The following example compares the `COL1` column of `TAB1` with national character set string `NCHAR literal`:

```
SELECT * FROM tab1 WHERE col1 = N'NCHAR literal';
```

NVARCHAR2 Datatype

The NVARCHAR2 datatype specifies a variable-length national character set character string. When you create a table with an NVARCHAR2 column, you supply the maximum number of characters or bytes it can hold. Oracle subsequently stores each value in the column exactly as you specify it, provided the value does not exceed the column's maximum length.

The column's maximum length is determined by the national character set definition. Width specifications of character datatype NVARCHAR2 refer to the number of characters if the national character set is fixed width and refer to the number of bytes if the national character set is variable width. The maximum column size allowed is 4000 bytes. For fixed-width, multibyte character sets, the maximum length of a column allowed is the number of characters that fit into no more than 4000 bytes.

The following statement creates a table with one NVARCHAR2 column of 2000 characters in length (stored as 4000 bytes, because each character takes two bytes) using JA16EUCFIXED as the national character set:

```
CREATE TABLE tab1 (col1 NVARCHAR2(2000));
```

VARCHAR2 Datatype

The VARCHAR2 datatype specifies a variable-length character string. When you create a VARCHAR2 column, you supply the maximum number of bytes of data that it can hold. Oracle subsequently stores each value in the column exactly as you specify it, provided the value does not exceed the column's maximum length. If you try to insert a value that exceeds the specified length, Oracle returns an error.

You must specify a maximum length for a VARCHAR2 column. This maximum must be at least 1 byte, although the actual length of the string stored is permitted to be zero. The maximum length of VARCHAR2 data is 4000 bytes. Oracle compares VARCHAR2 values using nonpadded comparison semantics. For information on comparison semantics, see "Datatype Comparison Rules" on page 2-27.

VARCHAR Datatype

The VARCHAR datatype is currently synonymous with the VARCHAR2 datatype. Oracle recommends that you use VARCHAR2 rather than VARCHAR. In the future, VARCHAR might be defined as a separate datatype used for variable-length character strings compared with different comparison semantics.

NUMBER Datatype

The NUMBER datatype stores zero, positive, and negative fixed and floating-point numbers with magnitudes between 1.0×10^{-130} and $9.9...9 \times 10^{125}$ (38 nines followed by 88 zeroes) with 38 digits of precision. If you specify an arithmetic expression whose value has a magnitude greater than or equal to 1.0×10^{126}, Oracle returns an error.

Specify a fixed-point number using the following form:

```
NUMBER(p,s)
```

where:

p is the *precision*, or the total number of digits. Oracle guarantees the portability of numbers with precision ranging from 1 to 38.

s is the *scale*, or the number of digits to the right of the decimal point. The scale can range from -84 to 127.

Specify an integer using the following form:

`NUMBER(p)` is a fixed-point number with precision p and scale 0. This is equivalent to NUMBER(p,0).

Specify a floating-point number using the following form:

`NUMBER` is a floating-point number with decimal precision 38. Note that a scale value is not applicable for floating-point numbers. (See "Floating-Point Numbers" on page 2-15 for more information.)

Scale and Precision

Specify the scale and precision of a fixed-point number column for extra integrity checking on input. Specifying scale and precision does not force all values to a fixed length. If a value exceeds the precision, Oracle returns an error. If a value exceeds the scale, Oracle rounds it.

The following examples show how Oracle stores data using different precisions and scales.

Actual Data	Specified As	Stored As
7456123.89	NUMBER	7456123.89
7456123.89	NUMBER(9)	7456124
7456123.89	NUMBER(9,2)	7456123.89
7456123.89	NUMBER(9,1)	7456123.9
7456123.89	NUMBER(6)	exceeds precision
7456123.89	NUMBER(7,-2)	7456100
7456123.89	NUMBER(-7,2)	exceeds precision

Negative Scale

If the scale is negative, the actual data is rounded to the specified number of places to the left of the decimal point. For example, a specification of (10,-2) means to round to hundreds.

Scale Greater than Precision

You can specify a scale that is greater than precision, although it is uncommon. In this case, the precision specifies the maximum number of digits to the right of the decimal point. As with all number datatypes, if the value exceeds the precision, Oracle returns an error message. If the value exceeds the scale, Oracle rounds the value. For example, a column defined as NUMBER(4,5) requires a zero for the first digit after the decimal point and rounds all values past the fifth digit after the decimal point. The following examples show the effects of a scale greater than precision:

Actual Data	Specified As	Stored As
.01234	NUMBER(4,5)	.01234
.00012	NUMBER(4,5)	.00012
.000127	NUMBER(4,5)	.00013
.0000012	NUMBER(2,7)	.0000012
.00000123	NUMBER(2,7)	.0000012

Floating-Point Numbers

Oracle allows you to specify floating-point numbers, which can have a decimal point anywhere from the first to the last digit or can have no decimal point at all. A scale value is not applicable to floating-point numbers, because the number of digits that can appear after the decimal point is not restricted.

You can specify floating-point numbers with the form discussed in "NUMBER Datatype" on page 2-13. Oracle also supports the ANSI datatype FLOAT. You can specify this datatype using one of these syntactic forms:

`FLOAT` specifies a floating-point number with decimal precision 38, or binary precision 126.

`FLOAT(`*b*`)` specifies a floating-point number with binary precision *b*. The precision *b* can range from 1 to 126. To convert from binary to decimal precision, multiply *b* by 0.30103. To convert from decimal to binary precision, multiply the decimal precision by 3.32193. The maximum of 126 digits of binary precision is roughly equivalent to 38 digits of decimal precision.

LONG Datatype

LONG columns store variable-length character strings containing up to 2 gigabytes, or 2^{31}-1 bytes. LONG columns have many of the characteristics of VARCHAR2 columns. You can use LONG columns to store long text strings. The length of LONG values may be limited by the memory available on your computer.

> **Note:** Oracle Corporation strongly recommends that you convert LONG columns to LOB columns. LOB columns are subject to far fewer restrictions than LONG columns. For more information, see "TO_LOB" on page 4-45.

You can reference LONG columns in SQL statements in these places:

- SELECT lists
- SET clauses of UPDATE statements
- VALUES clauses of INSERT statements

The use of LONG values is subject to some restrictions:

- A table cannot contain more than one LONG column.

- You cannot create an object type with a LONG attribute.
- LONG columns cannot appear in integrity constraints (except for NULL and NOT NULL constraints).
- LONG columns cannot be indexed.
- A stored function cannot return a LONG value.
- Within a single SQL statement, all LONG columns, updated tables, and locked tables must be located on the same database.

LONG columns cannot appear in certain parts of SQL statements:

- WHERE clauses, GROUP BY clauses, ORDER BY clauses, or CONNECT BY clauses or with the DISTINCT operator in SELECT statements
- The UNIQUE operator of a SELECT statement
- The column list of a CREATE CLUSTER statement
- The CLUSTER clause of a CREATE MATERIALIZED VIEW statement
- SQL functions (such as SUBSTR or INSTR)
- Expressions or conditions
- SELECT lists of queries containing GROUP BY clauses
- SELECT lists of subqueries or queries combined by set operators
- SELECT lists of CREATE TABLE ... AS SELECT statements
- SELECT lists in subqueries in INSERT statements

Triggers can use the LONG datatype in the following manner:

- A SQL statement within a trigger can insert data into a LONG column.
- If data from a LONG column can be converted to a constrained datatype (such as CHAR and VARCHAR2), a LONG column can be referenced in a SQL statement within a trigger.
- Variables in triggers cannot be declared using the LONG datatype.
- :NEW and :OLD cannot be used with LONG columns.

You can use the Oracle Call Interface functions to retrieve a portion of a LONG value from the database. See *Oracle Call Interface Programmer's Guide*.

DATE Datatype

The DATE datatype stores date and time information. Although date and time information can be represented in both CHAR and NUMBER datatypes, the DATE datatype has special associated properties. For each DATE value, Oracle stores the following information: century, year, month, day, hour, minute, and second.

To specify a date value, you must convert a character or numeric value to a date value with the TO_DATE function. Oracle automatically converts character values that are in the default date format into date values when they are used in date expressions. The default date format is specified by the initialization parameter NLS_DATE_FORMAT and is a string such as 'DD-MON-YY'. This example date format includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year.

If you specify a date value without a time component, the default time is 12:00:00 AM (midnight). If you specify a time value without a date, the default date is the first day of the current month.

The date function SYSDATE returns the current date and time. For information on the SYSDATE and TO_DATE functions and the default date format, see "Date Format Models" on page 2-40 and Chapter 4, "Functions".

Date Arithmetic

You can add and subtract number constants as well as other dates from dates. Oracle interprets number constants in arithmetic date expressions as numbers of days. For example, SYSDATE + 1 is tomorrow. SYSDATE - 7 is one week ago. SYSDATE + (10 / 1440) is ten minutes from now. Subtracting the HIREDATE column of the EMP table from SYSDATE returns the number of days since each employee was hired. You cannot multiply or divide DATE values.

Oracle provides functions for many common date operations. For example, the ADD_MONTHS function lets you add or subtract months from a date. The MONTHS_BETWEEN function returns the number of months between two dates. The fractional portion of the result represents that portion of a 31-day month. For more information on date functions, see "Date Functions" on page 4-4.

Because each date contains a time component, most results of date operations include a fraction. This fraction means a portion of one day. For example, 1.5 days is 36 hours.

Using Julian Dates

A Julian date is the number of days since January 1, 4712 BC. Julian dates allow continuous dating from a common reference. You can use the date format model "J" with date functions TO_DATE and TO_CHAR to convert between Oracle DATE values and their Julian equivalents.

Example This statement returns the Julian equivalent of January 1, 1997:

```
SELECT TO_CHAR(TO_DATE('01-01-1997', 'MM-DD-YYYY'),'J')
    FROM DUAL;

TO_CHAR
--------
2450450
```

For a description of the DUAL table, see "Selecting from the DUAL Table" on page 5-24.

RAW and LONG RAW Datatypes

The RAW and LONG RAW datatypes store data that is not to be interpreted (not explicitly converted when moving data between different systems) by Oracle. These datatypes are intended for binary data or byte strings. For example, you can use LONG RAW to store graphics, sound, documents, or arrays of binary data, for which the interpretation is dependent on the use.

Note: Oracle Corporation strongly recommends that you convert LONG RAW columns to binary LOB (BLOB) columns. LOB columns are subject to far fewer restrictions than LONG columns. For more information, see "TO_LOB" on page 4-45.

RAW is a variable-length datatype like VARCHAR2, except that Net8 (which connects user sessions to the instance) and the Import and Export utilities do not perform character conversion when transmitting RAW or LONG RAW data. In contrast, Net8 and Import/Export automatically convert CHAR, VARCHAR2, and LONG data from the database character set to the user session character set (which you can set with the NLS_LANGUAGE parameter of the ALTER SESSION statement), if the two character sets are different.

When Oracle automatically converts RAW or LONG RAW data to and from CHAR data, the binary data is represented in hexadecimal form, with one hexadecimal character representing every four bits of RAW data. For example, one byte of RAW data with bits 11001011 is displayed and entered as 'CB'.

Large Object (LOB) Datatypes

The built-in LOB datatypes BLOB, CLOB, and NCLOB (stored internally), and the BFILE (stored externally), can store large and unstructured data such as text, image, video, and spatial data up to 4 gigabytes in size.

When creating a table, you can optionally specify different tablespace and storage characteristics for LOB columns or LOB object attributes from those specified for the table.

LOB columns contain LOB locators that can refer to out-of-line or in-line LOB values. Selecting a LOB from a table actually returns the LOB's locator and not the entire LOB value. The DBMS_LOB package and Oracle Call Interface (OCI) operations on LOBs are performed through these locators. For more information about these interfaces and LOBs, see *Oracle8i Supplied Packages Reference* and *Oracle Call Interface Programmer's Guide*. For information on creating temporary LOBs, see *Oracle8i Application Developer's Guide - Large Objects (LOBs)*.

LOBs are similar to LONG and LONG RAW types, but differ in the following ways:

- LOBs can be attributes of a user-defined datatype (object).
- The LOB locator is stored in the table column, either with or without the actual LOB value. BLOB, NCLOB, and CLOB values can be stored in separate tablespaces. BFILE data is stored in an external file on the server.
- When you access a LOB column, the locator is returned.
- A LOB can be up to 4 gigabytes in size. BFILE maximum size is operating system dependent, but cannot exceed 4 gigabytes.
- LOBs permit efficient, random, piece-wise access to and manipulation of data.
- You can define more than one LOB column in a table.
- With the exception of NCLOB, you can define one or more LOB attributes in an object.
- You can declare LOB bind variables.
- You can select LOB columns and LOB attributes.

- You can insert a new row or update an existing row that contains one or more LOB columns and / or an object with one or more LOB attributes. (You can set the internal LOB value to NULL, empty, or replace the entire LOB with data. You can set the BFILE to NULL or make it point to a different file.)
- You can update a LOB row / column intersection or a LOB attribute with another LOB row / column intersection or LOB attribute.
- You can delete a row containing a LOB column or LOB attribute and thereby also delete the LOB value. Note that for BFILEs, the actual operating system file is not deleted.

For more information, please see the discussion of LOB restrictions in *Oracle8i Application Developer's Guide - Large Objects (LOBs)*. For more information on converting LONG columns to LOB columns, see "TO_LOB" on page 4-45.

To access and populate rows of an internal LOB column (a LOB column stored in the database), use the INSERT statement first to initialize the internal LOB value to empty. Once the row is inserted, you can select the empty LOB and populate it using the DBMS_LOB package or the OCI.

The following example creates a table with LOB columns:

```
CREATE TABLE  person_table (name CHAR(40),
                            resume  CLOB,
                            picture BLOB)
  LOB (resume) STORE AS
               ( TABLESPACE resumes
                  STORAGE (INITIAL 5M NEXT 5M) );
```

BFILE Datatype

The BFILE datatype enables access to binary file LOBs that are stored in file systems outside the Oracle database. A BFILE column or attribute stores a BFILE locator, which serves as a pointer to a binary file on the server's file system. The locator maintains the directory alias and the filename. See "CREATE DIRECTORY" on page 7-264.

Binary file LOBs do not participate in transactions and are not recoverable. Rather, the underlying operating system provides file integrity and durability. The maximum file size supported is 4 gigabytes.

The database administrator must ensure that the file exists and that Oracle processes have operating system read permissions on the file.

The BFILE datatype allows read-only support of large binary files. You cannot modify or replicate such a file. Oracle provides APIs to access file data. The

primary interfaces that you use to access file data are the DBMS_LOB package and the OCI. For more information about LOBs, see *Oracle8i Application Developer's Guide - Large Objects (LOBs)* and *Oracle Call Interface Programmer's Guide*.

BLOB Datatype

The BLOB datatype stores unstructured binary large objects. BLOBs can be thought of as bitstreams with no character set semantics. BLOBs can store up to 4 gigabytes of binary data.

BLOBs have full transactional support. Changes made through SQL, the DBMS_LOB package, or the OCI participate fully in the transaction. BLOB value manipulations can be committed and rolled back. Note, however, that you cannot save a BLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

CLOB Datatype

The CLOB datatype stores single-byte character data. Both fixed-width and variable-width character sets are supported, and both use the CHAR database character set. CLOBs can store up to 4 gigabytes of character data.

CLOBs have full transactional support. Changes made through SQL, the DBMS_LOB package, or the OCI participate fully in the transaction. CLOB value manipulations can be committed and rolled back. Note, however, that you cannot save a CLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

NCLOB Datatype

The NCLOB datatype stores multibyte national character set character (NCHAR) data. Both fixed-width and variable-width character sets are supported. NCLOBs can store up to 4 gigabytes of character text data.

NCLOBs have full transactional support. Changes made through SQL, the DBMS_LOB package, or the OCI participate fully in the transaction. NCLOB value manipulations can be committed and rolled back. Note, however, that you cannot save an NCLOB locator in a PL/SQL or OCI variable in one transaction and then use it in another transaction or session.

ROWID Datatype

Each row in the database has an address. You can examine a row's address by querying the pseudocolumn ROWID. Values of this pseudocolumn are

hexadecimal strings representing the address of each row. These strings have the datatype ROWID. For more information on the ROWID pseudocolumn, see "Pseudocolumns" on page 2-51. You can also create tables and clusters that contain actual columns having the ROWID datatype. Oracle does not guarantee that the values of such columns are valid rowids.

Restricted Rowids

Beginning with Oracle8, Oracle SQL incorporated an extended format for rowids to efficiently support partitioned tables and indexes and tablespace-relative data block addresses (DBAs) without ambiguity.

Character values representing rowids in Oracle7 and earlier releases are called **restricted** rowids. Their format is as follows:

```
block.row.file
```

where:

block	is a hexadecimal string identifying the data block of the datafile containing the row. The length of this string depends on your operating system.
row	is a four-digit hexadecimal string identifying the row in the data block. The first row of the block has a digit of 0.
file	is a hexadecimal string identifying the database file containing the row. The first datafile has the number 1. The length of this string depends on your operating system.

Extended Rowids

The **extended** ROWID datatype stored in a user column includes the data in the restricted rowid plus a **data object number**. The data object number is an identification number assigned to every database segment. You can retrieve the data object number from data dictionary views USER_OBJECTS, DBA_OBJECTS, and ALL_OBJECTS. Objects that share the same segment (clustered tables in the same cluster, for example) have the same object number.

Extended rowids are not available directly. You can use a supplied package, DBMS_ROWID, to interpret extended rowid contents. The package functions extract and provide information that would be available directly from a restricted rowid, as well as information specific to extended rowids. For information on the functions available with the DBMS_ROWID package and how to use them, see *Oracle8i Supplied Packages Reference*.

Compatibility and Migration

The restricted form of a rowid is still supported in Oracle8*i* for backward compatibility, but all tables return rowids in the extended format. For information regarding compatibility and migration issues, see *Oracle8i Migration*.

UROWID Datatype

Each row in a database has an address (as discussed in "ROWID Datatype" on page 2-21). However, the rows of some tables have addresses that are not physical or permanent, or were not generated by Oracle. For example, the row addresses of index-organized tables are stored in index leaves, which can move. Rowids of foreign tables (such as DB2 tables accessed through a gateway) are not standard Oracle rowids.

Oracle uses "universal rowids" (**urowids**) to store the addresses of index-organized and foreign tables. Index-organized tables have logical urowids and foreign tables have foreign urowids. Both types of urowid are stored in the ROWID pseudocolumn (as are the physical rowids of heap-organized tables).

Oracle creates logical rowids based on a table's primary key. The logical rowids do not change as long as the primary key does not change. The ROWID pseudocolumn of an index-organized table has a datatype of UROWID. You can access this pseudocolumn as you would the ROWID pseudocolumn of a heap-organized (that is, using the SELECT ROWID statement). If you wish to store the rowids of an index-organized table, you can define a column of type UROWID for the table and retrieve the value of the ROWID pseudocolumn into that column.

Note: Heap-organized tables have physical rowids. Oracle Corporation does not recommend that you specify a column of datatype UROWID for a heap-organized table.

For more information on the UROWID datatype and how Oracle generates and manipulates universal rowids, see *Oracle8i Concepts* and *Oracle8i Tuning*.

ANSI, DB2, and SQL/DS Datatypes

SQL statements that create tables and clusters can also use ANSI datatypes and datatypes from IBM's products SQL/DS and DB2. Oracle recognizes the ANSI or IBM datatype name and records it as the name of the datatype of the column, and then stores the column's data in an Oracle datatype based on the conversions shown in Table 2–2 and Table 2–3.

Table 2–2 ANSI Datatypes Converted to Oracle Datatypes

ANSI SQL Datatype	Oracle Datatype
CHARACTER(*n*)	CHAR(*n*)
CHAR(*n*)	
CHARACTER VARYING(*n*)	VARCHAR(*n*)
CHAR VARYING(*n*)	
NATIONAL CHARACTER(*n*)	NCHAR(*n*)
NATIONAL CHAR(*n*)	
NCHAR(*n*)	
NATIONAL CHARACTER VARYING(*n*)	NVARCHAR2(*n*)
NATIONAL CHAR VARYING(*n*)	
NCHAR VARYING(*n*)	
NUMERIC(*p*,*s*)	NUMBER(*p*,*s*)
DECIMAL(*p*,*s*)[a]	
INTEGER	NUMBER(38)
INT	
SMALLINT	
FLOAT(*b*)[b]	NUMBER
DOUBLE PRECISION[c]	
REAL[d]	

[a]The NUMERIC and DECIMAL datatypes can specify only fixed-point numbers. For these datatypes, *s* defaults to 0.

[b]The FLOAT datatype is a floating-point number with a binary precision *b*. The default precision for this datatype is 126 binary, or 38 decimal.

[c]The DOUBLE PRECISION datatype is a floating-point number with binary precision 126.

[d]The REAL datatype is a floating-point number with a binary precision of 63, or 18 decimal.

Table 2–3 SQL/DS and DB2 Datatypes Converted to Oracle Datatypes

SQL/DS or DB2 Datatype	Oracle Datatype
CHARACTER(*n*)	CHAR(*n*)
VARCHAR(*n*)	VARCHAR(*n*)
LONG VARCHAR(*n*)	LONG
DECIMAL(*p,s*)[a]	NUMBER(*p,s*)
INTEGER	NUMBER(38)
SMALLINT	
FLOAT(*b*)[b]	NUMBER

[a]The DECIMAL datatype can specify only fixed-point numbers. For this datatype, *s* defaults to 0.

[b]The FLOAT datatype is a floating-point number with a binary precision *b*. This default precision for this datatype is 126 binary, or 38 decimal.

Do not define columns with these SQL / DS and DB2 datatypes, because they have no corresponding Oracle datatype:

- GRAPHIC
- LONG VARGRAPHIC
- VARGRAPHIC
- TIME
- TIMESTAMP

Note that data of type TIME and TIMESTAMP can also be expressed as Oracle DATE data.

User-Defined Type Categories

User-defined datatypes use Oracle built-in datatypes and other user-defined datatypes as the building blocks of types that model the structure and behavior of data in applications. For information about Oracle built-in datatypes, see *Oracle8i Concepts*. For information about creating user-defined types, see "CREATE TYPE" on page 7-411 and the "CREATE TYPE BODY" on page 7-421. For information about using user-defined types, see *Oracle8i Application Developer's Guide - Fundamentals*.

The sections that follow describe the various categories of user-defined types.

Object Types

Object types are abstractions of the real-world entities, such as purchase orders, that application programs deal with. An object type is a schema object with three kinds of components:

- A *name*, which identifies the object type uniquely within that schema.
- *Attributes*, which are built-in types or other user-defined types. Attributes model the structure of the real-world entity.
- *Methods*, which are functions or procedures written in PL/SQL and stored in the database, or written in a language like C or Java and stored externally. Methods implement operations the application can perform on the real-world entity.

REFs

An **object identifier** (OID) uniquely identifies an object and enables you to reference the object from other objects or from relational tables. A datatype category called REF represents such references. A REF is a container for an object identifier. REFs are pointers to objects.

When a REF value points to a nonexistent object, the REF is said to be "dangling". A dangling REF is different from a null REF. To determine whether a REF is dangling or not, use the predicate `IS [NOT] DANGLING`. For example, given table DEPT with column MGR whose type is a REF to type EMP_T:

```
SELECT t.mgr.name
   FROM dept t
   WHERE t.mgr IS NOT DANGLING;
```

Varrays

An array is an ordered set of data elements. All elements of a given array are of the same datatype. Each element has an **index**, which is a number corresponding to the element's position in the array.

The number of elements in an array is the size of the array. Oracle arrays are of variable size, which is why they are called **varrays**. You must specify a maximum size when you declare the array.

When you declare a varray, it does not allocate space. It defines a type, which you can use as:

- The datatype of a column of a relational table
- An object type attribute
- A PL/SQL variable, parameter, or function return type

Oracle normally stores an array object either in line (that is, as part of the row data) or out of line (in a LOB), depending on its size. However, if you specify separate storage characteristics for a varray, Oracle will store it out of line, regardless of its size (see the *varray_storage_clause* of "CREATE TABLE" on page 7-359).

Nested Tables

A nested table type models an unordered set of elements. The elements may be built-in types or user-defined types. You can view a nested table as a single-column table or, if the nested table is an object type, as a multicolumn table, with a column for each attribute of the object type.

A nested table definition does not allocate space. It defines a type, which you can use to declare:

- Columns of a relational table
- Object type attributes
- PL/SQL variables, parameters, and function return values

When a nested table appears as the type of a column in a relational table or as an attribute of the underlying object type of an object table, Oracle stores all of the nested table data in a single table, which it associates with the enclosing relational or object table.

Datatype Comparison Rules

This section describes how Oracle compares values of each datatype.

Number Values

A larger value is considered greater than a smaller one. All negative numbers are less than zero and all positive numbers. Thus, -1 is less than 100; -100 is less than -1.

Date Values

A later date is considered greater than an earlier one. For example, the date equivalent of '29-MAR-1997' is less than that of '05-JAN-1998' and '05-JAN-1998 1:35pm' is greater than '05-JAN-1998 10:09am'.

Character String Values

Character values are compared using one of these comparison rules:

- blank-padded comparison semantics
- nonpadded comparison semantics

The following sections explain these comparison semantics. The results of comparing two character values using different comparison semantics may vary. The table below shows the results of comparing five pairs of character values using each comparison semantic. Usually, the results of blank-padded and nonpadded comparisons are the same. The last comparison in the table illustrates the differences between the blank-padded and nonpadded comparison semantics.

Blank-Padded	Nonpadded
`'ab' > 'aa'`	`'ab' > 'aa'`
`'ab' > 'a  '`	`'ab' > 'a  '`
`'ab' > 'a'`	`'ab' > 'a'`
`'ab' = 'ab'`	`'ab' = 'ab'`
'a ' = 'a'	**'a ' > 'a'**

Blank-Padded Comparison Semantics If the two values have different lengths, Oracle first adds blanks to the end of the shorter one so their lengths are equal. Oracle then compares the values character by character up to the first character that differs. The value with the greater character in the first differing position is considered greater. If two values have no differing characters, then they are considered equal. This rule means that two values are equal if they differ only in the number of trailing blanks. Oracle uses blank-padded comparison semantics only when both values in the comparison are either expressions of datatype CHAR, NCHAR, text literals, or values returned by the USER function.

Nonpadded Comparison Semantics Oracle compares two values character by character up to the first character that differs. The value with the greater character in that position is considered greater. If two values of different length are identical up to the end of the shorter one, the longer value is considered greater. If two values of equal length have no differing characters, then the values are considered equal. Oracle uses nonpadded comparison semantics whenever one or both values in the comparison have the datatype VARCHAR2 or NVARCHAR2.

Single Characters

Oracle compares single characters according to their numeric values in the database character set. One character is greater than another if it has a greater numeric value than the other in the character set. Oracle considers blanks to be less than any character, which is true in most character sets.

These are some common character sets:

- 7-bit ASCII (American Standard Code for Information Interchange)
- EBCDIC Code (Extended Binary Coded Decimal Interchange Code) Page 500
- ISO 8859 / 1 (International Standards Organization)
- JEUC Japan Extended UNIX

Portions of the ASCII and EBCDIC character sets appear in Table 2–4 and Table 2–5. Note that uppercase and lowercase letters are not equivalent. Also, note that the numeric values for the characters of a character set may not match the linguistic sequence for a particular language.

Table 2–4 ASCII Character Set

Symbol	Decimal value	Symbol	Decimal value
blank	32	;	59
!	33	<	60
"	34	=	61
#	35	>	62
$	36	?	63
%	37	@	64
&	38	A-Z	65-90
’	39	[	91
(	40	\	92
)	41	]	93
*	42	^^	94
+	43	_	95
,	44	‘	96
-	45	a-z	97-122

Table 2–4 (Cont.) ASCII Character Set

Symbol	Decimal value	Symbol	Decimal value
.	46	{	123
/	47	|	124
0-9	48-57	}	125
:	58	~	126

Table 2–5 EBCDIC Character Set

Symbol	Decimal value	Symbol	Decimal value
blank	64	%	108
¢	74	_	109
.	75	>	110
<	76	?	111
(	77	:	122
+	78	#	123
|	79	@	124
&	80	’	125
!	90	=	126
$	91	"	127
*	92	a-i	129-137
)	93	j-r	145-153
;	94	s-z	162-169
ÿ	95	A-I	193-201
-	96	J-R	209-217
/	97	S-Z	226-233

Object Values

Object values are compared using one of two comparison functions: MAP and ORDER. Both functions compare object type instances, but they are quite different from one another. These functions must be specified as part of the object type.

For a description of MAP and ORDER methods and the values they return, see "CREATE TYPE" on page 7-411. See also *Oracle8i Application Developer's Guide - Fundamentals* for more information.

Varrays and Nested Tables

You cannot compare varrays and nested tables in Oracle8*i*.

Data Conversion

Generally an expression cannot contain values of different datatypes. For example, an expression cannot multiply 5 by 10 and then add 'JAMES'. However, Oracle supports both implicit and explicit conversion of values from one datatype to another.

Implicit Data Conversion

Oracle automatically converts a value from one datatype to another when such a conversion makes sense. Oracle performs conversions in these cases:

- When an INSERT or UPDATE statement assigns a value of one datatype to a column of another, Oracle converts the value to the datatype of the column.
- When you use a SQL function or operator with an argument with a datatype other than the one it accepts, Oracle converts the argument to the accepted datatype.
- When you use a comparison operator on values of different datatypes, Oracle converts one of the expressions to the datatype of the other.

Example 1 The text literal '10' has datatype CHAR. Oracle implicitly converts it to the NUMBER datatype if it appears in a numeric expression as in the following statement:

```
SELECT sal + '10'
    FROM emp;
```

Example 2 When a condition compares a character value and a NUMBER value, Oracle implicitly converts the character value to a NUMBER value, rather than converting the NUMBER value to a character value. In the following statement, Oracle implicitly converts '7936' to 7936:

```
SELECT ename
    FROM emp
    WHERE empno = '7936';
```

Example 3 In the following statement, Oracle implicitly converts '12-MAR-1993' to a DATE value using the default date format 'DD-MON-YYYY':

```
SELECT ename
    FROM emp
    WHERE hiredate = '12-MAR-1993';
```

Example 4 In the following statement, Oracle implicitly converts the text literal 'AAAAZ8AABAAABvlAAA' to a rowid value:

```
SELECT ename
    FROM emp
    WHERE ROWID = 'AAAAZ8AABAAABvlAAA';
```

Explicit Data Conversion

You can also explicitly specify datatype conversions using SQL conversion functions. Table 2–6 shows SQL functions that explicitly convert a value from one datatype to another.

Table 2–6 SQL Functions for Datatype Conversion

TO: / FROM:	CHAR	NUMBER	DATE	RAW	ROWID	LONG/ LONG RAW	LOB
CHAR	–	TO_NUMBER	TO_DATE	HEXTORAW	CHARTO-ROWID		
NUMBER	TO_CHAR	–	TO_DATE (number,'J')				
DATE	TO_CHAR	TO_CHAR (date,'J')	–				
RAW	RAWTOHEX			–			
ROWID	ROWID-TOCHAR				–		
LONG / LONG RAW						–	TO_LOB
LOB							–

For information on these functions, see "Conversion Functions" on page 4-4.

Note: You cannot specify LONG and LONG RAW values in cases in which Oracle can perform implicit datatype conversion. For example, LONG and LONG RAW values cannot appear in expressions with functions or operators. For information on the limitations on LONG and LONG RAW datatypes, see "LONG Datatype" on page 2-15.

Implicit vs. Explicit Data Conversion

Oracle recommends that you specify explicit conversions rather than rely on implicit or automatic conversions for these reasons:

- SQL statements are easier to understand when you use explicit datatype conversion functions.
- Automatic datatype conversion can have a negative impact on performance, especially if the datatype of a column value is converted to that of a constant rather than the other way around.
- Implicit conversion depends on the context in which it occurs and may not work the same way in every case.
- Algorithms for implicit conversion are subject to change across software releases and among Oracle products. Behavior of explicit conversions is more predictable.

Format Models

A **format model** is a character literal that describes the format of DATE or NUMBER data stored in a character string. You can use a format model as an argument of the TO_CHAR and TO_DATE functions:

- To specify the format for Oracle to use to return a value from the database
- To specify the format for a value you have specified for Oracle to store in the database

See "TO_CHAR (date conversion)" on page 4-43, "TO_CHAR (number conversion)" on page 4-43, and "TO_DATE" on page 4-45. Note that a format model does not change the internal representation of the value in the database.

This section describes how to use:

- Number format models

- Date format models
- Format model modifiers

Changing the Return Format

You can use a format model to specify the format for Oracle to use to return values from the database to you.

Example 1 The following statement selects the commission values of the employees in Department 30 and uses the TO_CHAR function to convert these commissions into character values with the format specified by the number format model '$9,990.99':

```
SELECT ename employee, TO_CHAR(comm, '$9,990.99') commission
   FROM emp
   WHERE deptno = 30;

EMPLOYEE   COMMISSION
---------- ----------
ALLEN         $300.00
WARD          $500.00
MARTIN      $1,400.00
BLAKE
TURNER          $0.00
JAMES
```

Because of this format model, Oracle returns commissions with leading dollar signs, commas every three digits, and two decimal places. Note that TO_CHAR returns null for all employees with null in the COMM column.

Example 2 The following statement selects the date on which each employee from Department 20 was hired and uses the TO_CHAR function to convert these dates to character strings with the format specified by the date format model 'fmMonth DD, YYYY':

```
 SELECT ename, TO_CHAR(Hiredate,'fmMonth DD, YYYY') hiredate
    FROM emp
    WHERE deptno = 20;

ENAME      HIREDATE
---------- ------------------
SMITH      December 17, 1980
JONES      April 2, 1981
```

```
SCOTT      April 19, 1987
ADAMS      May 23, 1987
FORD       December 3, 1981
LEWIS      October 23, 1997
```

With this format model, Oracle returns the hire dates (as specified by "fm" and discussed in "Format Model Modifiers" on page 2-46) without blank padding, two digits for the day, and the century included in the year.

Supplying the Correct Format

You can use format models to specify the format of a value that you are converting from one datatype to another datatype required for a column. When you insert or update a column value, the datatype of the value that you specify must correspond to the column's datatype. For example, a value that you insert into a DATE column must be a value of the DATE datatype or a character string in the default date format (Oracle implicitly converts character strings in the default date format to the DATE datatype). If the value is in another format, you must use the TO_DATE function to convert the value to the DATE datatype. You must also use a format model to specify the format of the character string.

Example The following statement updates BAKER's hire date using the TO_DATE function with the format mask 'YYYY MM DD' to convert the character string '1998 05 20' to a DATE value:

```
UPDATE emp
  SET hiredate = TO_DATE('1998 05 20','YYYY MM DD')
  WHERE ename = 'BLAKE';
```

Number Format Models

You can use number format models:

- In the TO_CHAR function to translate a value of NUMBER datatype to VARCHAR2 datatype
- In the TO_NUMBER function to translate a value of CHAR or VARCHAR2 datatype to NUMBER datatype

All number format models cause the number to be rounded to the specified number of significant digits. If a value has more significant digits to the left of the decimal place than are specified in the format, pound signs (#) replace the value. If a positive value is extremely large and cannot be represented in the specified format, then the infinity sign (~) replaces the value. Likewise, if a negative value is

extremely small and cannot be represented by the specified format, then the negative infinity sign replaces the value (-~). This event typically occurs when you are using TO_CHAR() with a restrictive number format string, causing a rounding operation.

Number Format Elements

A number format model is composed of one or more number format elements. Table 2–7 lists the elements of a number format model. Examples are shown in Table 2–8.

Negative return values automatically contain a leading negative sign and positive values automatically contain a leading space unless the format model contains the MI, S, or PR format element.

Table 2–7 Number Format Elements

Element	Example	Description
, (comma)	`9,999`	Returns a comma in the specified position. You can specify multiple commas in a number format model. **Restrictions:** ■ A comma element cannot begin a number format model. ■ A comma cannot appear to the right of a decimal character or period in a number format model.
. (period)	`99.99`	Returns a decimal point, which is a period (.) in the specified position. **Restriction:** You can specify only one period in a number format model.
$	`$9999`	Returns value with a leading dollar sign.
0	`0999`	Returns leading zeros.
	`9990`	Returns trailing zeros.
9	`9999`	Returns value with the specified number of digits with a leading space if positive or with a leading minus if negative. Leading zeros are blank, except for a zero value, which returns a zero for the integer part of the fixed-point number.
B	`B9999`	Returns blanks for the integer part of a fixed-point number when the integer part is zero (regardless of "0"s in the format model).
C	`C999`	Returns in the specified position the ISO currency symbol (the current value of the `NLS_ISO_CURRENCY` parameter).

Table 2–7 (Cont.) Number Format Elements

Element	Example	Description
D	99D99	Returns in the specified position the decimal character, which is the current value of the NLS_NUMERIC_CHARACTER parameter. The default is a period (.).
		Restriction: You can specify only one decimal character in a number format model.
EEEE	9.9EEEE	Returns a value using in scientific notation.
FM	FM90.9	Returns a value with no leading or trailing blanks.
G	9G999	Returns in the specified position the group separator (the current value of the NLS_NUMERIC_CHARACTER parameter). You can specify multiple group separators in a number format model.
		Restriction: A group separator cannot appear to the right of a decimal character or period in a number format model.
L	L999	Returns in the specified position the local currency symbol (the current value of the NLS_CURRENCY parameter).
MI	9999MI	Returns negative value with a trailing minus sign (-).
		Returns positive value with a trailing blank.
		Restriction: The MI format element can appear only in the last position of a number format model.
PR	9999PR	Returns negative value in <angle brackets>.
		Returns positive value with a leading and trailing blank.
		Restriction: The PR format element can appear only in the last position of a number format model.
RN	RN	Returns a value as Roman numerals in uppercase.
rn	rn	Returns a value as Roman numerals in lowercase.
		Value can be an integer between 1 and 3999.
S	S9999	Returns negative value with a leading minus sign (-).
		Returns positive value with a leading plus sign (+).
	9999S	Returns negative value with a trailing minus sign (-).
		Returns positive value with a trailing plus sign (+).
		Restriction: The S format element can appear only in the first or last position of a number format model.

Table 2–7 (Cont.) Number Format Elements

Element	Example	Description
TM	`TM`	"Text minimum". Returns (in decimal output) the smallest number of characters possible. This element is case-insensitive. The default is TM9, which returns the number in fixed notation unless the output exceeds 64 characters. If output exceeds 64 characters, Oracle automatically returns the number in scientific notation. **Restrictions**: ■ You cannot precede this element with any other element. ■ You can follow this element only with 9 or E (only one) or e (only one).
U	`U9999`	Returns in the specified position the "Euro" (or other) dual currency symbol (the current value of the `NLS_DUAL_CURRENCY` parameter).
V	`999V99`	Returns a value multiplied by 10^n (and if necessary, round it up), where *n* is the number of 9's after the "V".
X	`XXXX` `xxxx`	Returns the hexadecimal value of the specified number of digits. If the specified number is not an integer, Oracle rounds it to an integer. **Restrictions**: ■ This element accepts only positive values or 0. Negative values return an error. ■ You can precede this element only with 0 (which returns leading zeroes) or FM. Any other elements return an error. If you specify neither 0 nor FM with X, the return always has 1 leading blank.

The values of some formats are determined by the value of initialization parameters. For such formats, you can specify the characters returned by these format elements implicitly using the initialization parameter NLS_TERRITORY. For information on these parameters, see *Oracle8i Reference* and *Oracle8i National Language Support Guide*.

You can change the default date format for your session with the ALTER SESSION statement. For information on changing the settings of these parameters, see "ALTER SESSION" on page 7-78.

Example Table 2–8 shows the results of the following query for different values of *number* and '*fmt*':

```
SELECT TO_CHAR(number, 'fmt')
  FROM DUAL;
```

Table 2–8 Results of Example Number Conversions

number	'fmt'	Result
-1234567890	9999999999S	'1234567890-'
0	99.99	' .00'
+0.1	99.99	' 0.10'
-0.2	99.99	' -.20'
0	90.99	' 0.00'
+0.1	90.99	' 0.10'
-0.2	90.99	' -0.20'
0	9999	' 0'
1	9999	' 1'
0	B9999	' '
1	B9999	' 1'
0	B90.99	' '
+123.456	999.999	' 123.456'
-123.456	999.999	'-123.456'
+123.456	FM999.009	'123.456'
+123.456	9.9EEEE	' 1.2E+02'
+1E+123	9.9EEEE	' 1.0E+123'
+123.456	FM9.9EEEE	'1.23E+02'
+123.45	FM999.009	'123.45'
+123.0	FM999.009	'123.00'
+123.45	L999.99	' $123.45'
+123.45	FML99.99	'$123.45'
+1234567890	9999999999S	'1234567890+'

Date Format Models

You can use date format models:

- In the TO_CHAR function to translate a DATE value that is in a format other than the default date format
- In the TO_DATE function to translate a character value that is in a format other than the default date format

Default Date Format

The default date format is specified either explicitly with the initialization parameter NLS_DATE_FORMAT or implicitly with the initialization parameter NLS_TERRITORY. For information on these parameters, see *Oracle8i Reference*.

You can change the default date format for your session with the ALTER SESSION statement. For information, see "ALTER SESSION" on page 7-78.

Maximum Length

The total length of a date format model cannot exceed 22 characters.

Date Format Elements

A date format model is composed of one or more date format elements as listed in Table 2–9.

- For input format models, format items cannot appear twice, and format items that represent similar information cannot be combined. For example, you cannot use 'SYYYY' and 'BC' in the same format string.
- Some of the date format elements cannot be used in the TO_DATE function, as noted in Table 2–9.

Capitalization of Date Format Elements Capitalization in a spelled-out word, abbreviation, or Roman numeral follows capitalization in the corresponding format element. For example, the date format model 'DAY' produces capitalized words like 'MONDAY'; 'Day' produces 'Monday'; and 'day' produces 'monday'.

Punctuation and Character Literals in Date Format Models You can also include these characters in a date format model:

- punctuation such as hyphens, slashes, commas, periods, and colons
- character literals, enclosed in double quotation marks

These characters appear in the return value in the same location as they appear in the format model.

Table 2–9 Date Format Elements

Element	Specify in TO_DATE?	Meaning
- / , . ; : 'text'	Yes	Punctuation and quoted text is reproduced in the result.
AD A.D.	Yes	AD indicator with or without periods.
AM A.M.	Yes	Meridian indicator with or without periods.
BC B.C.	Yes	BC indicator with or without periods.
CC SCC	No	One greater than the first two digits of a four-digit year; "S" prefixes BC dates with "-". For example, '20' from '1900'.
D	Yes	Day of week (1-7).
DAY	Yes	Name of day, padded with blanks to length of 9 characters.
DD	Yes	Day of month (1-31).
DDD	Yes	Day of year (1-366).
DY	Yes	Abbreviated name of day.
E	Yes	Abbreviated era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
EE	Yes	Full era name (Japanese Imperial, ROC Official, and Thai Buddha calendars).
HH	Yes	Hour of day (1-12).
HH12	Yes	Hour of day (1-12).
HH24	Yes	Hour of day (0-23).
IW	No	Week of year (1-52 or 1-53) based on the ISO standard.

Table 2–9 (Cont.) Date Format Elements

Element	Specify in TO_DATE?	Meaning
IYY IY I	No	Last 3, 2, or 1 digit(s) of ISO year.
IYYY	No	4-digit year based on the ISO standard.
J	Yes	Julian day; the number of days since January 1, 4712 BC. Number specified with 'J' must be integers.
MI	Yes	Minute (0-59).
MM	Yes	Two-digit numeric abbreviation of month (01-12; JAN = 01)
MON	Yes	Abbreviated name of month.
MONTH	Yes	Name of month, padded with blanks to length of 9 characters.
PM P.M.	No	Meridian indicator with or without periods.
Q	No	Quarter of year (1, 2, 3, 4; JAN-MAR = 1)
RM	Yes	Roman numeral month (I-XII; JAN = I).
RR	Yes	Given a year with 2 digits: ■ Returns a year in the next century if the year is <50 and the last 2 digits of the current year are >=50. ■ Returns a year in the preceding century if the year is >=50 and the last 2 digits of the current year are <50.
RRRR	Yes	Round year. Accepts either 4-digit or 2-digit input. If 2-digit, provides the same return as RR. If you don't want this functionality, enter the 4-digit year.
SS	Yes	Second (0-59).
SSSSS	Yes	Seconds past midnight (0-86399).
WW	No	Week of year (1-53) where week 1 starts on the first day of the year and continues to the seventh day of the year.
W	No	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.

Table 2–9 (Cont.) Date Format Elements

Element	Specify in TO_DATE?	Meaning
Y,YYY	Yes	Year with comma in this position.
YEAR SYEAR	No	Year, spelled out. "S" prefixes BC dates with "-".
YYYY SYYYY	Yes	4-digit year. "S" prefixes BC dates with "-".
YYY YY Y	Yes	Last 3, 2, or 1 digit(s) of year.

Oracle returns an error if an alphanumeric character is found in the date string where punctuation character is found in the format string. For example:

```
TO_CHAR (TO_DATE('0297','MM/YY'), 'MM/YY')
```

returns an error.

Date Format Elements and National Language Support

The functionality of some date format elements depends on the country and language in which you are using Oracle. For example, these date format elements return spelled values:

- MONTH
- MON
- DAY
- DY
- BC or AD or B.C. or A.D.
- AM or PM or A.M or P.M.

The language in which these values are returned is specified either explicitly with the initialization parameter NLS_DATE_LANGUAGE or implicitly with the initialization parameter NLS_LANGUAGE. The values returned by the YEAR and SYEAR date format elements are always in English.

The date format element D returns the number of the day of the week (1-7). The day of the week that is numbered 1 is specified implicitly by the initialization parameter NLS_TERRITORY.

For information on national language support initialization parameters, see *Oracle8i Reference* and *Oracle8i National Language Support Guide*.

ISO Standard Date Format Elements

Oracle calculates the values returned by the date format elements IYYY, IYY, IY, I, and IW according to the ISO standard. For information on the differences between these values and those returned by the date format elements YYYY, YYY, YY, Y, and WW, see the discussion of national language support in *Oracle8i National Language Support Guide*.

The RR Date Format Element

The RR date format element is similar to the YY date format element, but it provides additional flexibility for storing date values in other centuries. The RR date format element allows you to store 21st century dates in the 20th century by specifying only the last two digits of the year. It will also allow you to store 20th century dates in the 21st century in the same way if necessary.

If you use the TO_DATE function with the YY date format element, the date value returned is always in the current century. If you use the RR date format element instead, the century of the return value varies according to the specified two-digit year and the last two digits of the current year. Table 2–10 summarizes the behavior of the RR date format element.

Table 2–10 The RR Date Element Format

		If the specified two-digit year is	
		0 - 49	**50 - 99**
If the last two digits of the current year are:	**0-49**	The return date is in the current century.	The return date is in the preceding century.
	50-99	The return date is in the next century.	The return date is in the current century.

The following examples demonstrate the behavior of the RR date format element.

Example 1 Assume these queries are issued between 1950 and 1999:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR') ,'YYYY') "Year"
```

```
     FROM DUAL;

Year
----
1998

SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR') ,'YYYY') "Year";
     FROM DUAL;

Year
----
2017
```

Example 2 Assume these queries are issued between 2000 and 2049:

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR') ,'YYYY') "Year";
  FROM DUAL;

Year
----
1998

SELECT TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR') ,'YYYY') "Year";
     FROM DUAL;

Year
----
2017
```

Note that the queries return the same values regardless of whether they are issued before or after the year 2000. The RR date format element allows you to write SQL statements that will return the same values after the turn of the century.

Date Format Element Suffixes

Table 2–11 lists suffixes that can be added to date format elements:

Table 2–11 Date Format Element Suffixes

Suffix	Meaning	Example Element	Example Value
TH	Ordinal Number	DDTH	4TH
SP	Spelled Number	DDSP	FOUR
SPTH or THSP	Spelled, ordinal number	DDSPTH	FOURTH

Table 2–11 Date Format Element Suffixes

Suffix	Meaning	Example Element	Example Value

Restrictions:

- When you add one of these suffixes to a date format element, the return value is always in English.
- Date suffixes are valid only on output. You cannot use them to insert a date into the database.

Format Model Modifiers

The FM and FX modifiers, used in format models in the TO_CHAR function, control blank padding and exact format checking.

A modifier can appear in a format model more than once. In such a case, each subsequent occurrence toggles the effects of the modifier. Its effects are enabled for the portion of the model following its first occurrence, and then disabled for the portion following its second, and then reenabled for the portion following its third, and so on.

FM "Fill mode". This modifier suppresses blank padding in the return value of the TO_CHAR function:

- In a date format element of a TO_CHAR function, this modifier suppresses blanks in subsequent character elements (such as MONTH) and suppresses leading zeroes for subsequent number elements (such as MI) in a date format model. Without FM, the result of a character element is always right padded with blanks to a fixed length, and leading zeroes are always returned for a number element. With FM, because there is no blank padding, the length of the return value may vary.
- In a number format element of a TO_CHAR function, this modifier suppresses blanks added to the left of the number, so that the result is left-justified in the output buffer. Without FM, the result is always right-justified in the buffer, resulting in blank-padding to the left of the number.

FX "Format exact". This modifier specifies exact matching for the character argument and date format model of a TO_DATE function:

- Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.
- The character argument cannot have extra blanks. Without FX, Oracle ignores extra blanks.

- Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without FX, numbers in the character argument can omit leading zeroes.

  When FX is enabled, you can disable this check for leading zeroes by using the FM modifier as well.

If any portion of the character argument violates any of these conditions, Oracle returns an error message.

Example 1 The following statement uses a date format model to return a character expression:

```
SELECT TO_CHAR(SYSDATE, 'fmDDTH')||' of '||TO_CHAR
   (SYSDATE, 'Month')||', '||TO_CHAR(SYSDATE, 'YYYY') "Ides"
    FROM DUAL;

Ides
------------------
3RD of April, 1998
```

Note that the statement above also uses the FM modifier. If FM is omitted, the month is blank-padded to nine characters:

```
SELECT TO_CHAR(SYSDATE, 'DDTH')||' of '||
   TO_CHAR(Month, YYYY') "Ides"
   FROM DUAL;

Ides
-----------------------
03RD of April    , 1998
```

Example 2 The following statement places a single quotation mark in the return value by using a date format model that includes two consecutive single quotation marks:

```
SELECT TO_CHAR(SYSDATE, 'fmDay')||''s Special') "Menu"
     FROM DUAL;

Menu
-----------------
Tuesday's Special
```

Two consecutive single quotation marks can be used for the same purpose within a character literal in a format model.

Example 3 Table 2–12 shows whether the following statement meets the matching conditions for different values of *char* and *'fmt'* using FX:

```
UPDATE table
  SET date_column = TO_DATE(char, 'fmt');
```

Table 2–12 Matching Character Data and Format Models with the FX Format Model Modifier

char	'fmt'	Match or Error?
'15/ JAN /1998'	'DD-MON-YYYY'	Match
' 15! JAN % /1998'	'DD-MON-YYYY'	Error
'15/JAN/1998'	'FXDD-MON-YYYY'	Error
'15-JAN-1998'	'FXDD-MON-YYYY'	Match
'1-JAN-1998'	'FXDD-MON-YYYY'	Error
'01-JAN-1998'	'FXDD-MON-YYYY'	Match
'1-JAN-1998'	'FXFMDD-MON-YYYY'	Match

String-to-Date Conversion Rules

The following additional formatting rules apply when converting string values to date values (*unless* you have used the FX or FXFM modifiers in the format model to control exact format checking):

- You can omit punctuation included in the format string from the date string if all the digits of the numerical format elements, including leading zeros, are specified. In other words, specify 02 and not 2 for two-digit format elements such as MM, DD, and YY.
- You can omit time fields found at the end of a format string from the date string.
- If a match fails between a date format element and the corresponding characters in the date string, Oracle attempts alternative format elements, as shown in Table 2–13.

Table 2–13 Oracle Format Matching

Original Format Element	Additional Format Elements to Try in Place of the Original
'MM'	'MON' and 'MONTH'

Table 2–13 Oracle Format Matching

Original Format Element	Additional Format Elements to Try in Place of the Original
'MON	'MONTH'
'MONTH'	'MON'
'YY'	'YYYY'
'RR'	'RRRR'

Nulls

If a column in a row has no value, then the column is said to be **null**, or to contain a null. Nulls can appear in columns of any datatype that are not restricted by NOT NULL or PRIMARY KEY integrity constraints. Use a null when the actual value is not known or when a value would not be meaningful.

Do not use null to represent a value of zero, because they are not equivalent. (Oracle currently treats a character value with a length of zero as null. However, this may not continue to be true in future releases, and Oracle recommends that you do not treat empty strings the same as NULLs.) Any arithmetic expression containing a null always evaluates to null. For example, null added to 10 is null. In fact, all operators (except concatenation) return null when given a null operand.

Nulls in SQL Functions

All scalar functions (except NVL and TRANSLATE) return null when given a null argument. You can use the NVL function to return a value when a null occurs. For example, the expression NVL(COMM,0) returns 0 if COMM is null or the value of COMM if it is not null.

Most aggregate functions ignore nulls. For example, consider a query that averages the five values 1000, null, null, null, and 2000. Such a query ignores the nulls and calculates the average to be (1000+2000)/2 = 1500.

Nulls with Comparison Operators

To test for nulls, use only the comparison operators IS NULL and IS NOT NULL. If you use any other operator with nulls and the result depends on the value of the null, the result is UNKNOWN. Because null represents a lack of data, a null cannot be equal or unequal to any value or to another null. However, Oracle considers two

nulls to be equal when evaluating a DECODE expression. For syntax and additional information, see "DECODE Expressions" on page 5-12.

Oracle also considers two nulls to be equal if they appear in compound keys. That is, Oracle considers identical two compound keys containing nulls if all the non-null components of the keys are equal.

Nulls in Conditions

A condition that evaluates to UNKNOWN acts almost like FALSE. For example, a SELECT statement with a condition in the WHERE clause that evaluates to UNKNOWN returns no rows. However, a condition evaluating to UNKNOWN differs from FALSE in that further operations on an UNKNOWN condition evaluation will evaluate to UNKNOWN. Thus, NOT FALSE evaluates to TRUE, but NOT UNKNOWN evaluates to UNKNOWN.

Table 2–14 shows examples of various evaluations involving nulls in conditions. If the conditions evaluating to UNKNOWN were used in a WHERE clause of a SELECT statement, then no rows would be returned for that query.

Table 2–14 Conditions Containing Nulls

If A is:	Condition	Evaluates to:
10	a IS NULL	FALSE
10	a IS NOT NULL	TRUE
NULL	a IS NULL	TRUE
NULL	a IS NOT NULL	FALSE
10	a = NULL	UNKNOWN
10	a != NULL	UNKNOWN
NULL	a = NULL	UNKNOWN
NULL	a != NULL	UNKNOWN
NULL	a = 10	UNKNOWN
NULL	a != 10	UNKNOWN

For the truth tables showing the results of logical expressions containing nulls, see Table 3–6 on page 3-11, as well as Table 3–7 and Table 3–8.

Pseudocolumns

A **pseudocolumn** behaves like a table column, but is not actually stored in the table. You can select from pseudocolumns, but you cannot insert, update, or delete their values. This section describes these pseudocolumns:

- CURRVAL and NEXTVAL
- LEVEL
- ROWID
- ROWNUM

CURRVAL and NEXTVAL

A **sequence** is a schema object that can generate unique sequential values. These values are often used for primary and unique keys. You can refer to sequence values in SQL statements with these pseudocolumns:

CURRVAL	returns the current value of a sequence.
NEXTVAL	increments the sequence and returns the next value.

You must qualify CURRVAL and NEXTVAL with the name of the sequence:

```
sequence.CURRVAL
sequence.NEXTVAL
```

To refer to the current or next value of a sequence in the schema of another user, you must have been granted either SELECT object privilege on the sequence or SELECT ANY SEQUENCE system privilege, and you must qualify the sequence with the schema containing it:

```
schema.sequence.CURRVAL
schema.sequence.NEXTVAL
```

To refer to the value of a sequence on a remote database, you must qualify the sequence with a complete or partial name of a database link:

```
schema.sequence.CURRVAL@dblink
schema.sequence.NEXTVAL@dblink
```

For more information on referring to database links, see "Referring to Objects in Remote Databases" on page 2-74.

Where to Use Sequence Values

You can use CURRVAL and NEXTVAL in:

- The SELECT list of a SELECT statement that is not contained in a subquery, snapshot, or view
- The SELECT list of a subquery in an INSERT statement
- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

You *cannot* use CURRVAL and NEXTVAL:

- A subquery in a DELETE, SELECT, or UPDATE statement
- A view's query or snapshot's query
- A SELECT statement with the DISTINCT operator
- A SELECT statement with a GROUP BY clause or ORDER BY clause
- A SELECT statement that is combined with another SELECT statement with the UNION, INTERSECT, or MINUS set operator
- The WHERE clause of a SELECT statement
- DEFAULT value of a column in a CREATE TABLE or ALTER TABLE statement
- The condition of a CHECK constraint

Also, within a single SQL statement that uses CURVAL or NEXTVAL, all referenced LONG columns, updated tables, and locked tables must be located on the same database.

How to Use Sequence Values

When you create a sequence, you can define its initial value and the increment between its values. The first reference to NEXTVAL returns the sequence's initial value. Subsequent references to NEXTVAL increment the sequence value by the defined increment and return the new value. Any reference to CURRVAL always returns the sequence's current value, which is the value returned by the last reference to NEXTVAL. Note that before you use CURRVAL for a sequence in your session, you must first initialize the sequence with NEXTVAL.

Within a single SQL statement, Oracle will increment the sequence only once. If a statement contains more than one reference to NEXTVAL for a sequence, Oracle increments the sequence once and returns the same value for all occurrences of NEXTVAL. If a statement contains references to both CURRVAL and NEXTVAL,

Oracle increments the sequence and returns the same value for both CURRVAL and NEXTVAL regardless of their order within the statement.

A sequence can be accessed by many users concurrently with no waiting or locking. For information on sequences, see "CREATE SEQUENCE" on page 7-350.

Example 1 This example selects the current value of the employee sequence:

```
SELECT empseq.currval
    FROM DUAL;
```

Example 2 This example increments the employee sequence and uses its value for a new employee inserted into the employee table:

```
INSERT INTO emp
    VALUES (empseq.nextval, 'LEWIS', 'CLERK',
              7902, SYSDATE, 1200, NULL, 20);
```

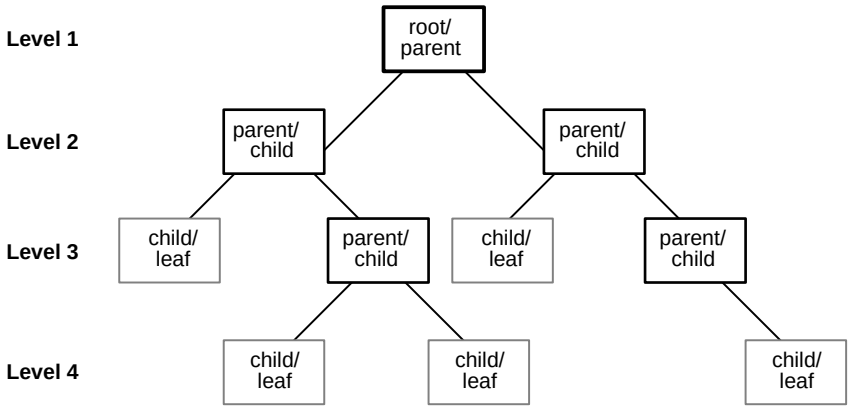
Example 3 This example adds a new order with the next order number to the master order table. It then adds suborders with this number to the detail order table:

```
INSERT INTO master_order(orderno, customer, orderdate)
    VALUES (orderseq.nextval, 'Al''s Auto Shop', SYSDATE);

INSERT INTO detail_order (orderno, part, quantity)
    VALUES (orderseq.currval, 'SPARKPLUG', 4);

INSERT INTO detail_order (orderno, part, quantity)
    VALUES (orderseq.currval, 'FUEL PUMP', 1);

INSERT INTO detail_order (orderno, part, quantity)
    VALUES (orderseq.currval, 'TAILPIPE', 2);
```

LEVEL

For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root node, 2 for a child of a root, and so on. A *root node* is the highest node within an inverted tree. A *child node* is any nonroot node. A *parent node* is any node that has children. A *leaf node* is any node without children. Figure 2–2 shows the nodes of an inverted tree with their LEVEL values.

Figure 2–2 Hierarchical Tree



To define a hierarchical relationship in a query, you must use the START WITH and CONNECT BY clauses. For more information on using the LEVEL pseudocolumn, see "SELECT and Subqueries" on page 7-541.

ROWID

For each row in the database, the ROWID pseudocolumn returns a row's address. Oracle8*i* rowid values contain information necessary to locate a row:

- the data object number of the object
- which data block in the datafile
- which row in the data block (first row is 0)
- which datafile (first file is 1). The file number is relative to the tablespace.

Usually, a rowid value uniquely identifies a row in the database. However, rows in different tables that are stored together in the same cluster can have the same rowid.

Values of the ROWID pseudocolumn have the datatype ROWID or UROWID. For more information, see "ROWID Datatype" on page 2-21 and "UROWID Datatype" on page 2-23.

Rowid values have several important uses:

- They are the fastest way to access a single row.
- They can show you how a table's rows are stored.

- They are unique identifiers for rows in a table.

You should not use ROWID as a table's primary key. If you delete and reinsert a row with the Import and Export utilities, for example, its rowid may change. If you delete a row, Oracle may reassign its rowid to a new row inserted later.

Although you can use the ROWID pseudocolumn in the SELECT and WHERE clause of a query, these pseudocolumn values are not actually stored in the database. You cannot insert, update, or delete a value of the ROWID pseudocolumn.

Example This statement selects the address of all rows that contain data for employees in department 20:

```
SELECT ROWID, ename
   FROM emp
   WHERE deptno = 20;

ROWID              ENAME
------------------ ----------
AAAAqYAABAAAEPvAAA SMITH
AAAAqYAABAAAEPvAAD JONES
AAAAqYAABAAAEPvAAH SCOTT
AAAAqYAABAAAEPvAAK ADAMS
AAAAqYAABAAAEPvAAM FORD
```

ROWNUM

For each row returned by a query, the ROWNUM pseudocolumn returns a number indicating the order in which Oracle selects the row from a table or set of joined rows. The first row selected has a ROWNUM of 1, the second has 2, and so on.

You can use ROWNUM to limit the number of rows returned by a query, as in this example:

```
SELECT * FROM emp WHERE ROWNUM < 10;
```

If an ORDER BY clause follows ROWNUM in the same subquery, the rows will be reordered by the ORDER BY clause. The results can vary depending on the way the rows are accessed. For example, if the ORDER BY clause causes Oracle to use an index to access the data, Oracle may retrieve the rows in a different order than without the index. Therefore, the following statement will not have the same effect as the preceding example:

```
SELECT * FROM emp WHERE ROWNUM < 11 ORDER BY empno;
```

If you embed the ORDER BY clause in a subquery and place the ROWNUM condition in the top-level query, you can force the ROWNUM condition to be applied after the ordering of the rows. For example, the following query returns the 10 smallest employee numbers. This is sometimes referred to as a "top-N query":

```
SELECT * FROM
   (SELECT empno FROM emp ORDER BY empno)
   WHERE ROWNUM < 11;
```

In the preceding example, the ROWNUM values are those of the top-level SELECT statement, so they are generated after the rows have already been ordered by EMPNO in the subquery. For more information about top-N queries, see *Oracle8i Application Developer's Guide - Fundamentals*.

Conditions testing for ROWNUM values *greater than* a positive integer are always false. For example, this query returns no rows:

```
SELECT * FROM emp
    WHERE ROWNUM > 1;
```

The first row fetched is assigned a ROWNUM of 1 and makes the condition false. The second row to be fetched is now the first row and is also assigned a ROWNUM of 1 and makes the condition false. All rows subsequently fail to satisfy the condition, so no rows are returned.

You can also use ROWNUM to assign unique values to each row of a table, as in this example:

```
UPDATE tabx
    SET col1 = ROWNUM;
```

Note: Using ROWNUM in a query can affect view optimization. For more information, see *Oracle8i Concepts*.

Comments

You can associate comments with SQL statements and schema objects.

Comments Within SQL Statements

Comments within SQL statements do not affect the statement execution, but they may make your application easier for you to read and maintain. You may want to

include a comment in a statement that describes the statement's purpose within your application.

A comment can appear between any keywords, parameters, or punctuation marks in a statement. You can include a comment in a statement using either of these means:

- Begin the comment with a slash and an asterisk (/*). Proceed with the text of the comment. This text can span multiple lines. End the comment with an asterisk and a slash (*/). The opening and terminating characters need not be separated from the text by a space or a line break.
- Begin the comment with -- (two hyphens). Proceed with the text of the comment. This text cannot extend to a new line. End the comment with a line break.

A SQL statement can contain multiple comments of both styles. The text of a comment can contain any printable characters in your database character set.

> **Note:** You cannot use these styles of comments between SQL statements in a SQL script. Use the SQL*Plus REMARK command for this purpose. For information on these statements, see *SQL*Plus User's Guide and Reference*.

Example These statements contain many comments:

```
SELECT ename, sal + NVL(comm, 0), job, loc
/* Select all employees whose compensation is
greater than that of Jones.*/
  FROM emp, dept
       /*The DEPT table is used to get the department name.*/
  WHERE emp.deptno = dept.deptno
    AND sal + NVL(comm,0) >   /* Subquery:       */
   (SELECT sal + NLV(comm,0)
                             /* total compensation is sal + comm */
      FROM emp
      WHERE ename = 'JONES');

SELECT ename,                     -- select the name
    sal + NVL(comm, 0),           -- total compensation
    job,                          -- job
    loc                           -- and city containing the office
  FROM emp,                       -- of all employees
       dept
```

```
WHERE emp.deptno = dept.deptno
  AND sal + NVL(comm, 0) >     -- whose compensation
                               -- is greater than
    (SELECT sal + NVL(comm,0)  -- the compensation
  FROM emp
  WHERE ename = 'JONES');       -- of Jones.
```

Comments on Schema Objects

You can associate a comment with a table, view, snapshot, or column using the COMMENT command described in Chapter 7, "SQL Statements". Comments associated with schema objects are stored in the data dictionary.

Hints

You can use comments in a SQL statement to pass instructions, or *hints*, to the Oracle optimizer. The optimizer uses these hints as suggestions for choosing an execution plan for the statement.

A statement block can have only one comment containing hints, and that comment must follow the SELECT, UPDATE, INSERT, or DELETE keyword. The syntax below shows hints contained in both styles of comments that Oracle supports within a statement block.

```
{DELETE|INSERT|SELECT|UPDATE} /*+ hint [text] [hint[text]]... */
```

or

```
{DELETE|INSERT|SELECT|UPDATE} --+ hint [text] [hint[text]]...
```

where

DELETE INSERT SELECT UPDATE	is a DELETE, INSERT, SELECT, or UPDATE keyword that begins a statement block. Comments containing hints can appear only after these keywords.
+	is a plus sign that causes Oracle to interpret the comment as a list of hints. The plus sign must follow immediately after the comment delimiter (no space is permitted).

hint is one of the hints discussed in this section and in *Oracle8i Tuning*. The space between the plus sign and the hint is optional. If the comment contains multiple hints, separate the hints by at least one space.

text is other commenting text that can be interspersed with the hints.

Table 2–15 lists hint syntax and descriptions. For more information on hints, see *Oracle8i Tuning* and *Oracle8i Concepts*.

Table 2–15 Hint Syntax and Descriptions

Hint Syntax	Description
Optimization Approaches and Goals	
`/*+ ALL_ROWS */`	Explicitly chooses the cost-based approach to optimize a statement block with a goal of best throughput (that is, minimum total resource consumption).
`/*+ CHOOSE */`	Causes the optimizer to choose between the rule-based approach and the cost-based approach for a SQL statement based on the presence of statistics for the tables accessed by the statement.
`/*+ FIRST_ROWS */`	Explicitly chooses the cost-based approach to optimize a statement block with a goal of best response time (minimum resource usage to return first row).
`/*+ RULE */`	Explicitly chooses rule-based optimization for a statement block.
Access Methods	
`/*+ AND_EQUAL(table index) */`	Explicitly chooses an execution plan that uses an access path that merges the scans on several single-column indexes.
`/*+ CLUSTER(table) */`	Explicitly chooses a cluster scan to access the specified table.
`/*+ FULL(table) */`	Explicitly chooses a full table scan for the specified table.
`/*+ HASH(table) */`	Explicitly chooses a hash scan to access the specified table.
`/*+ HASH_AJ(table) */`	Transforms a NOT IN subquery into a hash anti-join to access the specified table.
`/*+ HASH_SJ(table) */`	Transforms a NOT IN subquery into a hash semi-join to access the specified table.

Table 2–15 (Cont.) Hint Syntax and Descriptions

Hint Syntax	Description
`/*+ INDEX(table index) */`	Explicitly chooses an index scan for the specified table.
`/*+ INDEX_ASC(table index) */`	Explicitly chooses an ascending-range index scan for the specified table.
`/*+ INDEX_COMBINE(table index) */`	If no indexes are given as arguments for the INDEX_ COMBINE hint, the optimizer uses whatever Boolean combination of bitmap indexes has the best cost estimate. If particular indexes are given as arguments, the optimizer tries to use some Boolean combination of those particular bitmap indexes.
`/*+ INDEX_DESC(table index) */`	Explicitly chooses a descending-range index scan for the specified table.
`/*+ INDEX_FFS(table index) */`	Causes a fast full index scan to be performed rather than a full table scan.
`/*+ MERGE_AJ(table) */`	Transforms a NOT IN subquery into a merge anti-join to access the specified table.
`/*+ MERGE_SJ(table) */`	Transforms a correlated EXISTS subquery into a merge semi-join to access the specified table.
`/*+ NO_EXPAND */`	Prevents the optimizer from considering OR expansion for queries having OR or IN conditions in the WHERE clause.
`/*+ NO_INDEX(table index) */`	Instructs the optimizer not to consider a scan on the specified index or indexes. If no indexes are specified, the optimizer does not consider a scan on any index defined on the table.
`/*+ NOREWRITE */`	Disables query rewrite for the query block, overriding a TRUE setting of the QUERY_REWRITE_ENABLED parameter.
`/*+ ORDERED_PREDICATES */`	Forces the optimizer to preserve the order of predicate evaluation (except predicates used in index keys), as specified in the WHERE clause of SELECT statements.
`/*+ REWRITE (view [,...]) */`	Enforces query rewrite. If you specify a view list and the list contains an eligible materialized view, Oracle will use that view regardless of the cost. No views outside of the list are considered. If you do not specify a view list, Oracle will search for an eligible materialized view and always use it regardless of the cost.
`/*+ ROWID(table) */`	Explicitly chooses a table scan by rowid for the specified table.
`/*+ USE_CONCAT */`	Forces combined OR conditions in the WHERE clause of a query to be transformed into a compound query using the UNION ALL set operator.

Table 2–15 (Cont.) Hint Syntax and Descriptions

Hint Syntax	Description
Join Orders	
`/*+ ORDERED */`	Causes Oracle to join tables in the order in which they appear in the FROM clause.
`/*+ STAR */`	Forces the large table to be joined last using a nested-loops join on the index.
Join Operations	
`/*+ DRIVING_SITE(table) */`	Forces query execution to be done at a different site from that selected by Oracle.
`/*+ USE_HASH(table) */`	Causes Oracle to join each specified table with another row source with a hash join.
`/*+ USE_MERGE(table) */`	Causes Oracle to join each specified table with another row source with a sort-merge join.
`/*+ USE_NL(table) */`	Causes Oracle to join each specified table to another row source with a nested-loops join using the specified table as the inner table.
Parallel Execution	
Note: Oracle ignores parallel hints on a temporary table. For more information on temporary tables, see "CREATE TABLE" on page 7-359 and *Oracle8i Concepts*.	
`/*+ APPEND */` `/*+ NOAPPEND */`	Specifies that data is simply appended (or not) to a table; existing free space is not used. Use these hints only following the INSERT keyword.
`/*+ NOPARALLEL(table) */`	Disables parallel scanning of a table, even if the table was created with a PARALLEL clause. **Restriction:** You cannot parallelize a query involving a nested table.

Table 2–15 (Cont.) Hint Syntax and Descriptions

Hint Syntax	Description
`/*+ PARALLEL(table)` `/*+ PARALLEL(table, integer) */`	Lets you specify parallel execution of DML and queries on the table; *integer* specifies the desired degree of parallelism, which is the number of parallel threads that can be used for the operation. Each parallel thread may use one or two parallel execution servers. If you do not specify *integer*, Oracle computes a value using the PARALLEL_THREADS_PER_CPU parameter. If no parallel hint is specified, Oracle uses the existing degree of parallelism for the table. DELETE, INSERT, and UPDATE operations are considered for parallelization only if the session is in a PARALLEL DML enabled mode. (Use ALTER SESSION ENABLE PARALLEL DML to enter this mode.)
`/*+ PARALLEL_INDEX`	Allows you to parallelize fast full index scans for partitioned and nonpartitioned indexes that have the PARALLEL attribute.
`/*+ PQ_DISTRIBUTE` `(table, outer_distribution, inner_distribution) */`	Specifies how rows of joined tables should be distributed between producer and consumer query servers. The four possible distribution methods are NONE, HASH, BROADCAST, and PARTITION. However, only a subset of the combinations of outer and inner distributions are valid. For the permitted combinations of distributions for the outer and inner join tables, see *Oracle8i Tuning*.
`/*+ NOPARALLEL_INDEX */`	Overrides a PARALLEL attribute setting on an index.
Other Hints	
`/*+ CACHE */`	Specifies that the blocks retrieved for the table in the hint are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed.
`/*+ NOCACHE */`	Specifies that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed.
`/*+ MERGE(table) */`	Causes Oracle to evaluate complex views or subqueries before the surrounding query.
`/*+ NO_MERGE(table) */`	Causes Oracle not to merge mergeable views.
`/*+ PUSH_JOIN_PRED(table) */`	Causes the optimizer to evaluate, on a cost basis, whether to push individual join predicates into the view.

Table 2–15 (Cont.) Hint Syntax and Descriptions

Hint Syntax	Description
`/*+ NO_PUSH_JOIN_PRED(table) */`	Prevents pushing of a join predicate into the view.
`/*+ PUSH_SUBQ */`	Causes nonmerged subqueries to be evaluated at the earliest possible place in the execution plan.
`/*+ STAR_TRANSFORMATION */`	Makes the optimizer use the best plan in which the transformation has been used.

Database Objects

Oracle recognizes objects that are associated with a particular schema and objects that are not associated with a particular schema, as described in the sections that follow.

Schema Objects

A **schema** is a collection of logical structures of data, or schema objects. A schema is owned by a database user and has the same name as that user. Each user owns a single schema. Schema objects can be created and manipulated with SQL and include the following types of objects:

- clusters
- database links
- database triggers
- dimensions
- external procedure libraries
- index-organized tables
- indexes
- indextypes
- materialized views / snapshots
- materialized view logs / snapshot logs
- object tables
- object types
- object views

- operators
- packages
- sequences
- stored functions
- stored procedures
- synonyms
- tables
- views

Nonschema Objects

Other types of objects are also stored in the database and can be created and manipulated with SQL but are not contained in a schema:

- contexts
- directories
- profiles
- roles
- rollback segments
- tablespaces
- users

In this reference, each type of object is briefly defined in Chapter 7, "SQL Statements", in the section describing the statement that creates the database object. These statements begin with the keyword CREATE. For example, for the definition of a cluster, see "CREATE CLUSTER" on page 7-236. For an overview of database objects, see *Oracle8i Concepts*.

You must provide names for most types of schema objects when you create them. These names must follow the rules listed in the following sections.

Parts of Schema Objects

Some schema objects are made up of parts that you can or must name, such as:

- columns in a table or view

- index and table partitions and subpartitions
- integrity constraints on a table
- packaged procedures, packaged stored functions, and other objects stored within a package

Partitioned Tables and Indexes

Tables and indexes can be partitioned. When partitioned, these schema objects consist of a number of parts called *partitions*, all of which have the same logical attributes. For example, all partitions in a table share the same column and constraint definitions, and all partitions in an index share the same index columns.

When you partition a table or index using the range method, you specify a maximum value for the partitioning key column(s) for each partition. When you partition a table or index using the hash method, you instruct Oracle to distribute the rows of the table into partitions based on a system-defined hash function on the partitioning key column(s). When you partition a table or index using the composite-partitioning method, you specify ranges for the partitions, and Oracle distributes the rows in each partition into one or more hash subpartitions based on a hash function. Each subpartition of a table or index partitioned using the composite method has the same logical attributes.

Partition-Extended and Subpartition-Extended Table Names

Partition-extended and subpartition-extended table names let you perform some partition-level and subpartition-level operations, such as deleting all rows from a partition or subpartition, on only one partition or subpartition. Without extended table names, such operations would require that you specify a predicate (WHERE clause). For range-partitioned tables, trying to phrase a partition-level operation with a predicate can be cumbersome, especially when the range partitioning key uses more than one column. For hash partitions and subpartitions, using a predicate is more difficult still, because these partitions and subpartitions are based on a system-defined hash function.

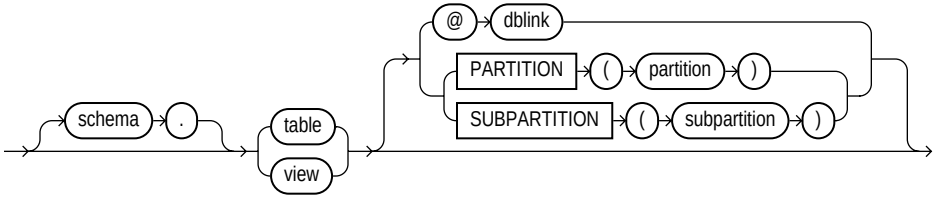
Partition-extended table names let you use partitions as if they were tables. An advantage of this method, which is most useful for range-partitioned tables, is that you can build partition-level access control mechanisms by granting (or revoking) privileges on these views to (or from) other users or roles.To use a partition as a table, create a view by selecting data from a single partition, and then use the view as a table.

You can specify partition-extended or subpartition-extended table names for the following DML statements:

- DELETE
- INSERT
- LOCK TABLE
- SELECT
- UPDATE

> **Note:** For application portability and ANSI syntax compliance, Oracle strongly recommends that you use views to insulate applications from this Oracle proprietary extension.

Syntax The basic syntax for using partition-extended and subpartition-extended table names is:



Restrictions Currently, the use of partition-extended and subpartition-extended table names has the following restrictions:

- No remote tables: A partition-extended or subpartition-extended table name cannot contain a database link (dblink) or a synonym that translates to a table with a dblink. To use remote partitions and subpartitions, create a view at the remote site that uses the extended table name syntax and then refer to the remote view.
- No direct PL/SQL support: A SQL statement using the extended table name syntax cannot be used in a PL/SQL block, although it can be used through dynamic SQL by using the DBMS_SQL package. To refer to a partition or subpartition within a PL/SQL block, use views that in turn use the extended table name syntax.

- No synonyms: A partition or subpartition extension must be specified with a base table. You cannot use synonyms, views, or any other objects.

Example In the following statement, SALES is a partitioned table with partition JAN97. You can create a view of the single partition JAN97, and then use it as if it were a table. This example deletes rows from the partition.

```
CREATE VIEW sales_jan97 AS
     SELECT * FROM sales PARTITION (jan97);
DELETE FROM sales_jan97 WHERE amount < 0;
```

Schema Object Names and Qualifiers

This section provides:

- rules for naming schema objects and schema object location qualifiers
- guidelines for naming schema objects and qualifiers

Schema Object Naming Rules

The following rules apply when naming schema objects:

1. Names must be from 1 to 30 characters long with these exceptions:
   - Names of databases are limited to 8 characters.
   - Names of database links can be as long as 128 characters.
2. Names cannot contain quotation marks.
3. Names are not case sensitive.
4. A name must begin with an alphabetic character from your database character set unless surrounded by double quotation marks.
5. Names can contain only alphanumeric characters from your database character set and the underscore (_), dollar sign ($), and pound sign (#). Oracle strongly discourages you from using $ and #. Names of database links can also contain periods (.) and "at" signs (@).

   If your database character set contains multibyte characters, Oracle recommends that each name for a user or a role contain at least one single-byte character.

Note: You cannot use special characters from European or Asian character sets in a database name, global database name, or database link names. For example, characters with an umlaut are not allowed.

---

6. A name cannot be an Oracle reserved word. Appendix C, "Oracle Reserved Words", lists all Oracle reserved words.

   Depending on the Oracle product you plan to use to access a database object, names might be further restricted by other product-specific reserved words. For a list of a product's reserved words, see the manual for the specific product, such as *PL/SQL User's Guide and Reference*.

7. Do not use the word DUAL as a name for an object or part. DUAL is the name of a dummy table.

8. The Oracle SQL language contains other words that have special meanings. These words include datatypes (see "Datatypes" on page 2-5), function names (see "SQL Functions" on page 4-1), and keywords (the uppercase words in SQL statements, such as DIMENSION, SEGMENT, ALLOCATE, DISABLE, and so forth). These words are not reserved. However, Oracle uses them internally. Therefore, if you use these words as names for objects and object parts, your SQL statements may be more difficult to read and may lead to unpredictable results.

   In particular, do not use words beginning with "SYS_" as schema object names, and do not use the names of SQL built-in functions for the names of schema objects or user-defined functions.

9. Within a namespace, no two objects can have the same name.

   Figure 2–3 shows the namespaces for schema objects. Each box is a namespace. Tables and views are in the same namespace. Therefore, a table and a view in the same schema cannot have the same name. However, tables and indexes are in different namespaces. Therefore, a table and an index in the same schema can have the same name.

   Each schema in the database has its own namespaces for the objects it contains. This means, for example, that two tables in different schemas are in different namespaces and can have the same name.

Figure 2–3 Namespaces for Schema Objects

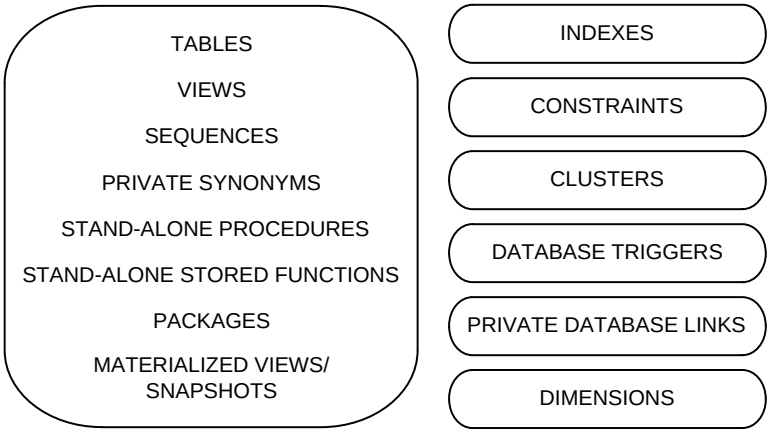
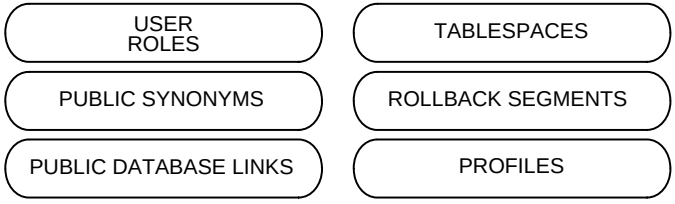


Figure 2–4 shows the namespaces for nonschema objects. Because the objects in these namespaces are not contained in schemas, these namespaces span the entire database.

Figure 2–4 Namespaces for Nonschema Objects



10. Columns in the same table or view cannot have the same name. However, columns in different tables or views can have the same name.

11. Procedures or functions contained in the same package can have the same name, provided that their arguments are not of the same number and datatypes. Creating multiple procedures or functions with the same name in the same package with different arguments is called *overloading* the procedure or function.

12. A name can be enclosed in double quotation marks. Such names can contain any combination of characters, including spaces, ignoring rules 3 through 7 in

this list. This exception is allowed for portability, but Oracle recommends that you do not break rules 3 through 7.

If you give a schema object a name enclosed in double quotation marks, you must use double quotation marks whenever you refer to the object.

Enclosing a name in double quotes allows it to:

- Contain spaces
- Be case sensitive
- Begin with a character other than an alphabetic character, such as a numeric character
- Contain characters other than alphanumeric characters and _, $, and #
- Be a reserved word

By enclosing names in double quotation marks, you can give the following names to different objects in the same namespace:

```
emp
"emp"
"Emp"
"EMP "
```

Note that Oracle interprets the following names the same, so they cannot be used for different objects in the same namespace:

```
emp
EMP
"EMP"
```

If you give a user or password a quoted name, the name cannot contain lowercase letters.

Database link names cannot be quoted.

Schema Object Naming Examples

The following examples are valid schema object names:

```
ename
horse
scott.hiredate
"EVEN THIS & THAT!"
a_very_long_and_valid_name
```

Although column aliases, table aliases, usernames, and passwords are not objects or parts of objects, they must also follow these naming rules with these exceptions:

- Column aliases and table aliases exist only for the execution of a single SQL statement and are not stored in the database, so rule 12 does not apply to them.
- Passwords do not have namespaces, so rule 9 does not apply to them.
- Do not use quotation marks to make usernames and passwords case sensitive. For additional rules for naming users and passwords, see "CREATE USER" on page 7-425.

Schema Object Naming Guidelines

Here are several helpful guidelines for naming objects and their parts:

- Use full, descriptive, pronounceable names (or well-known abbreviations).
- Use consistent naming rules.
- Use the same name to describe the same entity or attribute across tables.

When naming objects, balance the objective of keeping names short and easy to use with the objective of making names as descriptive as possible. When in doubt, choose the more descriptive name, because the objects in the database may be used by many people over a period of time. Your counterpart ten years from now may have difficulty understanding a database with a name like PMDD instead of PAYMENT_DUE_DATE.

Using consistent naming rules helps users understand the part that each table plays in your application. One such rule might be to begin the names of all tables belonging to the FINANCE application with FIN_.

Use the same names to describe the same things across tables. For example, the department number columns of the sample EMP and DEPT tables are both named DEPTNO.

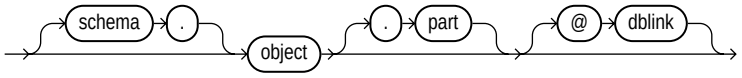
Referring to Schema Objects and Parts

This section tells you how to refer to schema objects and their parts in the context of a SQL statement. This section shows you:

- the general syntax for referring to an object
- how Oracle resolves a reference to an object

- how to refer to objects in schemas other than your own
- how to refer to objects in remote databases

The following diagram shows the general syntax for referring to an object or a part:



where:

object is the name of the object.

schema is the schema containing the object. The schema qualifier allows you to refer to an object in a schema other than your own. You must be granted privileges to refer to objects in other schemas. If you omit *schema*, Oracle assumes that you are referring to an object in your own schema.

Only schema objects can be qualified with *schema*. Schema objects are shown in Figure 2–3 on page 2-69. Nonschema objects, shown in Figure 2–4 on page 2-69, cannot be qualified with *schema* because they are not schema objects. (An exception is public synonyms, which can optionally be qualified with "PUBLIC". The quotation marks are required.)

part is a part of the object. This identifier allows you to refer to a part of a schema object, such as a column or a partition of a table. Not all types of objects have parts.

dblink applies only when you are using Oracle's distributed functionality. This is the name of the database containing the object. The *dblink* qualifier lets you refer to an object in a database other than your local database. If you omit *dblink*, Oracle assumes that you are referring to an object in your local database. Not all SQL statements allow you to access objects on remote databases.

You can include spaces around the periods separating the components of the reference to the object, but it is conventional to omit them.

How Oracle Resolves Schema Object References

When you refer to an object in a SQL statement, Oracle considers the context of the SQL statement and locates the object in the appropriate namespace. After locating the object, Oracle performs the statement's operation on the object. If the named object cannot be found in the appropriate namespace, Oracle returns an error.

The following example illustrates how Oracle resolves references to objects within SQL statements. Consider this statement that adds a row of data to a table identified by the name DEPT:

```
INSERT INTO dept
   VALUES (50, 'SUPPORT', 'PARIS');
```

Based on the context of the statement, Oracle determines that DEPT can be:

- a table in your own schema
- a view in your own schema
- a private synonym for a table or view
- a public synonym

Oracle always attempts to resolve an object reference within the namespaces in your own schema before considering namespaces outside your schema. In this example, Oracle attempts to resolve the name DEPT as follows:

1. First, Oracle attempts to locate the object in the namespace in your own schema containing tables, views, and private synonyms. If the object is a private synonym, Oracle locates the object for which the synonym stands. This object could be in your own schema, another schema, or on another database. The object could also be another synonym, in which case Oracle locates the object for which this synonym stands.
2. If the object is in the namespace, Oracle attempts to perform the statement on the object. In this example, Oracle attempts to add the row of data to DEPT. If the object is not of the correct type for the statement, Oracle returns an error. In this example, DEPT must be a table, view, or a private synonym resolving to a table or view. If DEPT is a sequence, Oracle returns an error.
3. If the object is not in any namespace searched in thus far, Oracle searches the namespace containing public synonyms (see Figure 2–4 on page 2-69). If the object is in that namespace, Oracle attempts to perform the statement on it. If the object is not of the correct type for the statement, Oracle returns an error. In this example, if DEPT is a public synonym for a sequence, Oracle returns an error.

Referring to Objects in Other Schemas

To refer to objects in schemas other than your own, prefix the object name with the schema name:

```
schema.object
```

For example, this statement drops the EMP table in the schema SCOTT:

```
DROP TABLE scott.emp
```

Referring to Objects in Remote Databases

To refer to objects in databases other than your local database, follow the object name with the name of the database link to that database. A database link is a schema object that causes Oracle to connect to a remote database to access an object there. This section tells you:

- How to create database links
- How to use database links in your SQL statements

Creating Database Links

You create a database link with the CREATE DATABASE LINK statement described in Chapter 7, "SQL Statements". The statement allows you to specify this information about the database link:

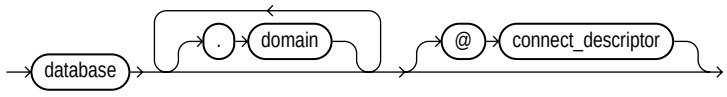
- The name of the database link
- The database connect string to access the remote database
- The username and password to connect to the remote database

Oracle stores this information in the data dictionary.

Database Link Names When you create a database link, you must specify its name. Database link names are different from names of other types of objects. They can be as long as 128 bytes and can contain periods (.) and the "at" sign (@).

The name that you give to a database link must correspond to the name of the database to which the database link refers and the location of that database in the hierarchy of database names. The following syntax diagram shows the form of the name of a database link:

dblink::=



where:

database	should specify *name* portion of the *global name* of the remote database to which the database link connects. This global name is stored in the data dictionary of the remote database; you can see this name in the GLOBAL_NAME view.
domain	should specify the *domain* portion of the global name of the remote database to which the database link connects. If you omit *domain* from the name of a database link, Oracle qualifies the database link name with the domain of your local database as it currently exists in the data dictionary.
connect_descriptor	allows you to further qualify a database link. Using connect descriptors, you can create multiple database links to the same database. For example, you can use connect descriptors to create multiple database links to different instances of the Oracle Parallel Server that access the same database.

The combination *database.domain* is sometimes called the "service name". For more information, see *Net8 Administrator's Guide*.

Username and Password Oracle uses the username and password to connect to the remote database. The username and password for a database link are optional.

Database Connect String The database connect string is the specification used by Net8 to access the remote database. For information on writing database connect strings, see the Net8 documentation for your specific network protocol. The database string for a database link is optional.

Referring to Database Links

Database links are available only if you are using Oracle's distributed functionality. When you issue a SQL statement that contains a database link, you can specify the database link name in one of these forms:

complete	is the complete database link name as stored in the data dictionary, including the *database*, *domain*, and optional *connect_descriptor* components.
partial	is the *database* and optional *connect_descriptor* components, but not the *domain* component.

Oracle performs these tasks before connecting to the remote database:

1. If the database link name specified in the statement is partial, Oracle expands the name to contain the domain of the local database as found in the global database name stored in the data dictionary. (You can see the current global database name in the GLOBAL_NAME data dictionary view.)
2. Oracle first searches for a private database link in your own schema with the same name as the database link in the statement. Then, if necessary, it searches for a public database link with the same name.
   - Oracle always determines the username and password from the first matching database link (either private or public). If the first matching database link has an associated username and password, Oracle uses it. If it does not have an associated username and password, Oracle uses your current username and password.
   - If the first matching database link has an associated database string, Oracle uses it. If not, Oracle searches for the next matching (public) database link. If no matching database link is found, or if no matching link has an associated database string, Oracle returns an error.
3. Oracle uses the database string to access the remote database. After accessing the remote database, if the value of the GLOBAL_NAMES parameter is TRUE, Oracle verifies that the *database.domain* portion of the database link name matches the complete global name of the remote database. If this condition is true, Oracle proceeds with the connection, using the username and password chosen in Step 2. If not, Oracle returns an error.
4. If the connection using the database string, username, and password is successful, Oracle attempts to access the specified object on the remote database using the rules for resolving object references and referring to objects in other schemas discussed earlier in this section.

You can disable the requirement that the *database.domain* portion of the database link name must match the complete global name of the remote database by setting to FALSE the initialization parameter GLOBAL_NAMES or the GLOBAL_NAMES parameter of the ALTER SYSTEM or ALTER SESSION statement.

For more information on remote name resolution, see *Oracle8i Distributed Database Systems*.

Referencing Object Type Attributes and Methods

To reference object type attributes or methods in a SQL statement, you must fully qualify the reference with a table alias. Consider the following example:

```
CREATE TYPE person AS OBJECT
   (ssno VARCHAR(20),
    name VARCHAR (10));

CREATE TABLE emptab (pinfo person);
```

In a SQL statement, reference to the SSNO attribute must be fully qualified using a table alias, as illustrated below:

```
SELECT e.pinfo.ssno FROM emptab e;

UPDATE emptab e SET e.pinfo.ssno = '510129980'
   WHERE e.pinfo.name = 'Mike';
```

To reference an object type's member method that does not accept arguments, you must provide "empty" parentheses. For example, assume that AGE is a method in the person type that does not take arguments. In order to call this method in a SQL statement, you must provide empty parentheses as shows in this example:

```
SELECT e.pinfo.age() FROM emptab e
   WHERE e.pinfo.name = 'Mike';
```

For more information, see the sections on user-defined datatypes in *Oracle8i Concepts*.

3

Operators

With affection beaming in one eye, and calculation shining out of the other.

Charles Dickens, *Martin Chuzzlewit*

An operator manipulates individual data items and returns a result. The data items are called *operands* or *arguments*. Operators are represented by special characters or by keywords. For example, the multiplication operator is represented by an asterisk (*) and the operator that tests for nulls is represented by the keywords IS NULL.

This chapter discusses the following topics:

Unary and Binary Operators

The two general classes of operators are:

unary — A unary operator operates on only one operand. A unary operator typically appears with its operand in this format:

```
operator operand
```

binary — A binary operator operates on two operands. A binary operator appears with its operands in this format:

```
operand1 operator operand2
```

Other operators with special formats accept more than two operands. If an operator is given a null operand, the result is always null. The only operator that does not follow this rule is concatenation (||).

Precedence

Precedence is the order in which Oracle evaluates different operators in the same expression. When evaluating an expression containing multiple operators, Oracle evaluates operators with higher precedence before evaluating those with lower precedence. Oracle evaluates operators with equal precedence from left to right within an expression.

Table 3–1 lists the levels of precedence among SQL operators from high to low. Operators listed on the same line have the same precedence.

Table 3–1 SQL Operator Precedence

Operator	Operation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	comparison
NOT	exponentiation, logical negation
AND	conjunction
OR	disjunction

Example In the following expression, multiplication has a higher precedence than addition, so Oracle first multiplies 2 by 3 and then adds the result to 1.

```
1+2*3
```

You can use parentheses in an expression to override operator precedence. Oracle evaluates expressions inside parentheses before evaluating those outside.

SQL also supports set operators (UNION, UNION ALL, INTERSECT, and MINUS), which combine sets of rows returned by queries, rather than individual data items. All set operators have equal precedence.

Arithmetic Operators

You can use an arithmetic operator in an expression to negate, add, subtract, multiply, and divide numeric values. The result of the operation is also a numeric value. Some of these operators are also used in date arithmetic. Table 3–2 lists arithmetic operators.

Table 3–2 Arithmetic Operators

Operator	Purpose	Example

+ -	Denotes a positive or negative expression. These are unary operators.	`SELECT * FROM orders`
		`WHERE qtysold = -1;`
		`SELECT * FROM emp`
		`WHERE -sal < 0;`

* /	Multiplies, divides. These are binary operators.	`UPDATE emp` `SET sal = sal * 1.1;`

+ -	Adds, subtracts. These are binary operators.	`SELECT sal + comm FROM emp`
		`WHERE SYSDATE - hiredate`
		`> 365;`

Do not use two consecutive minus signs (--) in arithmetic expressions to indicate double negation or the subtraction of a negative value. The characters -- are used to begin comments within SQL statements. You should separate consecutive minus signs with a space or a parenthesis. For more information on comments within SQL statements, see "Comments" on page 2-56.

Concatenation Operator

The concatenation operator manipulates character strings. Table 3–3 describes the concatenation operator.

Table 3–3 Concatenation Operator

Operator	Purpose	Example
||	Concatenates character strings.	`SELECT 'Name is ' || ename FROM emp;`

The result of concatenating two character strings is another character string. If both character strings are of datatype CHAR, the result has datatype CHAR and is limited to 2000 characters. If either string is of datatype VARCHAR2, the result has datatype VARCHAR2 and is limited to 4000 characters. Trailing blanks in character strings are preserved by concatenation, regardless of the strings' datatypes. For more information on the differences between the CHAR and VARCHAR2 datatypes, see "Character Datatypes" on page 2-10.

On most platforms, the concatenation operator is two solid vertical bars, as shown in Table 3–3. However, some IBM platforms use broken vertical bars for this operator. When moving SQL script files between systems having different character sets, such as between ASCII and EBCDIC, vertical bars might not be translated into the vertical bar required by the target Oracle environment. Oracle provides the CONCAT character function as an alternative to the vertical bar operator for cases when it is difficult or impossible to control translation performed by operating system or network utilities. Use this function in applications that will be moved between environments with differing character sets.

Although Oracle treats zero-length character strings as nulls, concatenating a zero-length character string with another operand always results in the other operand, so null can result only from the concatenation of two null strings. However, this may not continue to be true in future versions of Oracle. To concatenate an expression that might be null, use the NVL function to explicitly convert the expression to a zero-length string.

Example This example creates a table with both CHAR and VARCHAR2 columns, inserts values both with and without trailing blanks, and then selects these values and concatenates them. Note that for both CHAR and VARCHAR2 columns, the trailing blanks are preserved.

```
CREATE TABLE tab1 (col1 VARCHAR2(6), col2 CHAR(6),
        col3 VARCHAR2(6), col4 CHAR(6) );

Table created.

INSERT INTO tab1 (col1,  col2,     col3,     col4)
        VALUES   ('abc', 'def   ', 'ghi   ', 'jkl');
```

```
1 row created.

SELECT col1||col2||col3||col4 "Concatenation"
        FROM tab1;

Concatenation
------------------------
abcdef   ghi   jkl
```

Comparison Operators

Comparison operators compare one expression with another. The result of such a comparison can be TRUE, FALSE, or UNKNOWN. For information on conditions, see "Conditions" on page 5-13. Table 3–4 lists comparison operators.

Table 3–4 Comparison Operators

Operator	Purpose	Example
=	Equality test.	`SELECT * FROM emp WHERE sal = 1500;`
!= ^= < > ¬=	Inequality test. Some forms of the inequality operator may be unavailable on some platforms.	`SELECT * FROM emp WHERE sal != 1500;`
>	"Greater than" and "less than" tests.	`SELECT * FROM emp WHERE sal > 1500;`
<		`SELECT * FROM emp WHERE sal < 1500;`
>=	"Greater than or equal to" and "less than or equal to" tests.	`SELECT * FROM emp WHERE sal >= 1500;`
<=		`SELECT * FROM emp WHERE sal <= 1500;`
IN	"Equal to any member of" test. Equivalent to "= ANY".	`SELECT * FROM emp WHERE job IN ('CLERK','ANALYST');` `SELECT * FROM emp WHERE sal IN (SELECT sal FROM emp WHERE deptno = 30);`

Table 3–4 (Cont.) Comparison Operators

Operator	Purpose	Example
NOT IN	Equivalent to "!=ALL". Evaluates to FALSE if any member of the set is NULL.	SELECT * FROM emp WHERE sal NOT IN (SELECT sal FROM emp WHERE deptno = 30); SELECT * FROM emp WHERE job NOT IN ('CLERK', ANALYST');
ANY SOME	Compares a value to each value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to FALSE if the query returns no rows.	SELECT * FROM emp WHERE sal = ANY (SELECT sal FROM emp WHERE deptno = 30);
ALL	Compares a value to every value in a list or returned by a query. Must be preceded by =, !=, >, <, <=, >=. Evaluates to TRUE if the query returns no rows.	SELECT * FROM emp WHERE sal >= ALL (1400, 3000);
[NOT] BETWEEN x AND y	[Not] greater than or equal to *x* and less than or equal to *y*.	SELECT * FROM emp WHERE sal BETWEEN 2000 AND 3000;
EXISTS	TRUE if a subquery returns at least one row.	SELECT ename, deptno FROM dept WHERE EXISTS (SELECT * FROM emp WHERE dept.deptno = emp.deptno);
x [NOT] LIKE y [ESCAPE 'z']	TRUE if *x* does [not] match the pattern *y*. Within *y*, the character "%" matches any string of zero or more characters except null. The character "_" matches any single character. Any character, excepting percent (%) and underbar (_) may follow ESCAPE. A wildcard character is treated as a literal if preceded by the character designated as the escape character.	See "LIKE Operator" on page 3-7. SELECT * FROM tab1 WHERE col1 LIKE 'A_C/%E%' ESCAPE '/';

Table 3–4 (Cont.) Comparison Operators

Operator	Purpose	Example
`IS [NOT] NULL`	Tests for nulls. This is the only operator that you should use to test for nulls. See "Nulls" on page 2-49.	`SELECT ename, deptno FROM emp WHERE comm IS NULL;`

Additional information on the NOT IN and LIKE operators appears in the sections that follow.

NOT IN Operator

If any item in the list following a NOT IN operation is null, all rows evaluate to UNKNOWN (and no rows are returned). For example, the following statement returns the string 'TRUE' for each row:

```
SELECT 'TRUE'
    FROM emp
    WHERE deptno NOT IN (5,15);
```

However, the following statement returns no rows:

```
SELECT 'TRUE'
    FROM emp
    WHERE deptno NOT IN (5,15,null);
```

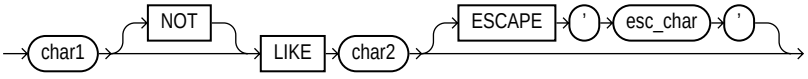
The above example returns no rows because the WHERE clause condition evaluates to:

```
deptno != 5 AND deptno != 15 AND deptno != null
```

Because all conditions that compare a null result in a null, the entire expression results in a null. This behavior can easily be overlooked, especially when the NOT IN operator references a subquery.

LIKE Operator

The LIKE operator is used in character string comparisons with pattern matching. The syntax for a condition using the LIKE operator is shown in this diagram:



where:

char1	is a value to be compared with a pattern. This value can have datatype CHAR or VARCHAR2.
NOT	logically inverts the result of the condition, returning FALSE if the condition evaluates to TRUE and TRUE if it evaluates to FALSE.
char2	is the pattern to which *char1* is compared. The pattern is a value of datatype CHAR or VARCHAR2 and can contain the special pattern matching characters % and _.
ESCAPE	identifies a single character as the escape character. The escape character can be used to cause Oracle to interpret % or _ literally, rather than as a special character. If you wish to search for strings containing an escape character, you must specify this character twice. For example, if the escape character is '/', to search for the string 'client/server', you must specify, 'client//server'.

Whereas the equal (=) operator exactly matches one character value to another, the LIKE operator matches a portion of one character value to another by searching the first value for the pattern specified by the second. Note that blank padding is **not** used for LIKE comparisons.

With the LIKE operator, you can compare a value to a pattern rather than to a constant. The pattern must appear after the LIKE keyword. For example, you can issue the following query to find the salaries of all employees with names beginning with 'SM':

```
SELECT sal
    FROM emp
    WHERE ename LIKE 'SM%';
```

The following query uses the = operator, rather than the LIKE operator, to find the salaries of all employees with the name 'SM%':

```
SELECT sal
    FROM emp
```

```
    WHERE ename = 'SM%';
```

The following query finds the salaries of all employees with the name 'SM%'. Oracle interprets 'SM%' as a text literal, rather than as a pattern, because it *precedes* the LIKE operator:

```
SELECT sal
    FROM emp
    WHERE 'SM%' LIKE ename;
```

Patterns typically use special characters that Oracle matches with different characters in the value:

- An underscore (_) in the pattern matches exactly one character (as opposed to one byte in a multibyte character set) in the value.
- A percent sign (%) in the pattern can match zero or more characters (as opposed to bytes in a multibyte character set) in the value. Note that the pattern '%' cannot match a null.

Case Sensitivity and Pattern Matching Case is significant in all conditions comparing character expressions including the LIKE and equality (=) operators. You can use the UPPER() function to perform a case-insensitive match, as in this condition:

```
UPPER(ename) LIKE 'SM%'
```

Pattern Matching on Indexed Columns When LIKE is used to search an indexed column for a pattern, Oracle can use the index to improve the statement's performance if the leading character in the pattern is not "%" or "_". In this case, Oracle can scan the index by this leading character. If the first character in the pattern is "%" or "_", the index cannot improve the query's performance because Oracle cannot scan the index.

Example 1 This condition is true for all ENAME values beginning with "MA":

```
ename LIKE 'MA%'
```

All of these ENAME values make the condition TRUE:

```
MARTIN, MA, MARK, MARY
```

Case is significant, so ENAME values beginning with "Ma," "ma," and "mA" make the condition FALSE.

Example 2 Consider this condition:

```
ename LIKE 'SMITH_'
```

This condition is true for these ENAME values:

```
SMITHE, SMITHY, SMITHS
```

This condition is false for 'SMITH', since the special character "_" must match exactly one character of the ENAME value.

ESCAPE Option You can include the actual characters "%" or "_" in the pattern by using the ESCAPE option. The ESCAPE option identifies the escape character. If the escape character appears in the pattern before the character "%" or "_" then Oracle interprets this character literally in the pattern, rather than as a special pattern matching character.

Example: To search for employees with the pattern 'A_B' in their name:

```
SELECT ename
    FROM emp
    WHERE ename LIKE '%A\_B%' ESCAPE '\';
```

The ESCAPE option identifies the backslash (\) as the escape character. In the pattern, the escape character precedes the underscore (_). This causes Oracle to interpret the underscore literally, rather than as a special pattern matching character.

Patterns Without % If a pattern does not contain the "%" character, the condition can be TRUE only if both operands have the same length.

Example: Consider the definition of this table and the values inserted into it:

```
CREATE TABLE freds (f CHAR(6), v VARCHAR2(6));
INSERT INTO freds VALUES ('FRED', 'FRED');
```

Because Oracle blank-pads CHAR values, the value of F is blank-padded to 6 bytes. V is not blank-padded and has length 4.

Logical Operators

A logical operator combines the results of two component conditions to produce a single result based on them or to invert the result of a single condition. Table 3–5 lists logical operators.

Table 3–5 Logical Operators

Operator	Function	Example
NOT	Returns TRUE if the following condition is FALSE. Returns FALSE if it is TRUE. If it is UNKNOWN, it remains UNKNOWN.	`SELECT * FROM emp WHERE NOT (job IS NULL);` `SELECT * FROM emp WHERE NOT (sal BETWEEN 1000 AND 2000);`
AND	Returns TRUE if both component conditions are TRUE. Returns FALSE if either is FALSE. Otherwise returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' AND deptno = 10;`
OR	Returns TRUE if either component condition is TRUE. Returns FALSE if both are FALSE. Otherwise returns UNKNOWN.	`SELECT * FROM emp WHERE job = 'CLERK' OR deptno = 10;`

For example, in the WHERE clause of the following SELECT statement, the AND logical operator is used to ensure that only those hired before 1984 and earning more than $1000 a month are returned:

```
SELECT *
    FROM emp
    WHERE hiredate < TO_DATE('01-JAN-1984', 'DD-MON-YYYY')
        AND sal > 1000;
```

NOT Operator

Table 3–6 shows the result of applying the NOT operator to a condition.

Table 3–6 NOT Truth Table

	TRUE	FALSE	UNKNOWN
NOT	FALSE	TRUE	UNKNOWN

AND Operator

Table 3–7 shows the results of combining two expressions with AND.

Table 3–7 AND Truth Table

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR Operator

Table 3–8 shows the results of combining two expressions with OR.

Table 3–8 OR Truth Table

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Set Operators

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries. Table 3–9 lists SQL set operators.

Table 3–9 Set Operators

Operator	Returns
UNION	All rows selected by either query.
UNION ALL	All rows selected by either query, including all duplicates.
INTERSECT	All distinct rows selected by both queries.
MINUS	All distinct rows selected by the first query but not the second.

All set operators have equal precedence. If a SQL statement contains multiple set operators, Oracle evaluates them from the left to right if no parentheses explicitly specify another order.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and datatype. If component queries select character data, the datatype of the return values are determined as follows:

- If both queries select values of datatype CHAR, the returned values have datatype CHAR.
- If either or both of the queries select values of datatype VARCHAR2, the returned values have datatype VARCHAR2.

Examples Consider these two queries and their results:

```
SELECT part
    FROM orders_list1;

PART
----------
SPARKPLUG
FUEL PUMP
FUEL PUMP
TAILPIPE

SELECT part
    FROM orders_list2;

PART
----------
CRANKSHAFT
TAILPIPE
TAILPIPE
```

The following examples combine the two query results with each of the set operators.

UNION Example The following statement combines the results with the UNION operator, which eliminates duplicate selected rows. This statement shows how datatype must match when columns do not exist in one or the other table:

```
SELECT part, partnum, to_date(null) date_in
    FROM orders_list1
UNION
SELECT part, to_date(null), date_in
    FROM orders_list2;

PART       PARTNUM DATE_IN
```

```
---------- ------- --------
SPARKPLUG  3323165
SPARKPLUG          10/24/98
FUEL PUMP  3323162
FUEL PUMP          12/24/99
TAILPIPE   1332999
TAILPIPE           01/01/01
CRANKSHAFT 9394991
CRANKSHAFT         09/12/02

SELECT part
    FROM orders_list1
UNION
SELECT part
    FROM orders_list2;

PART
----------
SPARKPLUG
FUEL PUMP
TAILPIPE
CRANKSHAFT
```

UNION ALL Example The following statement combines the results with the UNION ALL operator, which does not eliminate duplicate selected rows:

```
SELECT part
    FROM orders_list1
UNION ALL
SELECT part
    FROM orders_list2;

PART
----------
SPARKPLUG
FUEL PUMP
FUEL PUMP
TAILPIPE
CRANKSHAFT
TAILPIPE
TAILPIPE
```

Note that the UNION operator returns only distinct rows that appear in either result, while the UNION ALL operator returns all rows. A PART value that appears

multiple times in either or both queries (such as 'FUEL PUMP') is returned only once by the UNION operator, but multiple times by the UNION ALL operator.

INTERSECT Example The following statement combines the results with the INTERSECT operator, which returns only those rows returned by both queries:

```
SELECT part
    FROM orders_list1
INTERSECT
SELECT part
    FROM orders_list2;

PART
----------
TAILPIPE
```

MINUS Example The following statement combines results with the MINUS operator, which returns only rows returned by the first query but not by the second:

```
SELECT part
    FROM orders_list1
MINUS
SELECT part
    FROM orders_list2;

PART
----------
SPARKPLUG
FUEL PUMP
```

Other Built-In Operators

Table 3–10 lists other SQL operators.

Table 3–10 Other SQL Operators

Operator	Purpose	Example
(+)	Indicates that the preceding column is the outer join column in a join. See "Outer Joins" on page 5-22.	`SELECT ename, dname FROM emp, dept WHERE dept.deptno = emp.deptno(+);`
PRIOR	Evaluates the following expression for the parent row of the current row in a hierarchical, or tree-structured, query. In such a query, you must use this operator in the CONNECT BY clause to define the relationship between parent and child rows. You can also use this operator in other parts of a SELECT statement that performs a hierarchical query. The PRIOR operator is a unary operator and has the same precedence as the unary + and - arithmetic operators. See "Hierarchical Queries" on page 5-19.	`SELECT empno, ename, mgr FROM emp CONNECT BY PRIOR empno = mgr;`

User-Defined Operators

Like built-in operators, user-defined operators take a set of operands as input and return a result. However, you create them with the CREATE OPERATOR statement, and they are identified by names (e.g., MERGE). They reside in the same namespace as tables, views, types, and stand-alone functions.

Once you have defined a new operator, you can use it in SQL statements like any other built-in operator. For example, you can use user-defined operators in the select list of a SELECT statement, the condition of a WHERE clause, or in ORDER BY clauses and GROUP BY clauses. However, you must have EXECUTE privilege on the operator to do so, because it is a user-defined object.

For example, if you define an operator CONTAINS, which takes as input a text document and a keyword and returns 1 if the document contains the specified keyword, you can then write the following SQL query:

```
SELECT * FROM emp WHERE contains (resume, 'Oracle and UNIX') = 1;
```

For more information on user-defined operators, see "CREATE OPERATOR" on page 7-320 and *Oracle8i Data Cartridge Developer's Guide* .

4

Functions

Form ever follows function.

Louis Henri Sullivan, *The Tall Office Building Artistically Considered*

Functions are similar to operators in that they manipulate data items and return a result. Functions differ from operators in the format in which they appear with their arguments. This format allows them to operate on zero, one, two, or more arguments:

```
function(argument, argument, ...)
```

This chapter describes two types of functions:

- SQL Functions
- User-Defined Functions

SQL Functions

SQL functions are built into Oracle and are available for use in various appropriate SQL statements. Do not confuse SQL functions with user functions written in PL/SQL. User functions are described in "User-Defined Functions" on page 4-56. For information about functions used with Oracle interMedia, see *Oracle8i interMedia Audio, Image, and Video User's Guide and Reference.*

If you call a SQL function with an argument of a datatype other than the datatype expected by the SQL function, Oracle implicitly converts the argument to the expected datatype before performing the SQL function. See "Data Conversion" on page 2-31.

If you call a SQL function with a null argument, the SQL function automatically returns null. The only SQL functions that do not follow this rule are CONCAT, DECODE, DUMP, NVL, and REPLACE.

In the syntax diagrams for SQL functions, arguments are indicated by their datatypes, following the conventions described in "Syntax Diagrams and Notation" in the Preface of this reference. When the parameter "function" appears in SQL syntax, replace it with one of the functions described in this section. Functions are grouped by the datatypes of their arguments and their return values. The general syntax is as follows:

function::=

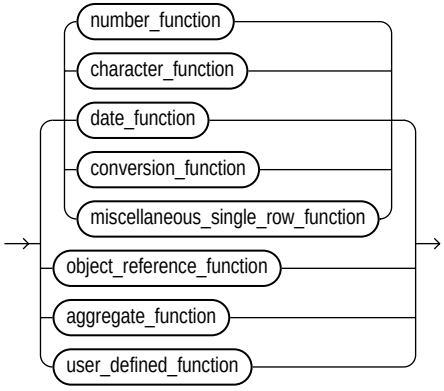


Table 4–1 lists the built-in SQL functions in each of the groups illustrated above except user-defined functions. All of the built-in SQL functions are then described in alphabetical order. User-defined functions are described at the end of this chapter.

Table 4–1 SQL Function Groups

Group	Functions		Description
Single-Row Functions	Single-row functions return a single result row for every row of a queried table or view. Single-row functions can appear in select lists (if the SELECT statement does not contain a GROUP BY clause), WHERE clauses, START WITH clauses, and CONNECT BY clauses.		
Number Functions	Number functions accept numeric input and return numeric values. Most of these functions return values that are accurate to 38 decimal digits. The transcendental functions COS, COSH, EXP, LN, LOG, SIN, SINH, SQRT, TAN, and TANH are accurate to 36 decimal digits. The transcendental functions ACOS, ASIN, ATAN, and ATAN2 are accurate to 30 decimal digits.		
	ABS	EXP	SIGN
	ACOS	FLOOR	SIN
	ADD_MONTHS	LN	SINH
	ATAN	LOG	SQRT
	ATAN2	MOD	TAN
	CEIL	POWER	TANH
	COS	ROUND (Number Function)	TRUNC (Number Function)
	COSH		
Character Functions returning character values	Character functions that return character values, unless otherwise noted, return values with the datatype VARCHAR2 and are limited in length to 4000 bytes. Functions that return values of datatype CHAR are limited in length to 2000 bytes. If the length of the return value exceeds the limit, Oracle truncates it and returns the result without an error message.		
	CHR	NLS_LOWER	SUBSTR
	CONCAT	NLSSORT	SUBSTRB
	INITCAP	NLS_UPPER	TRANSLATE
	LOWER	REPLACE	TRIM
	LPAD	RPAD	UPPER
	LTRIM	RTRIM	
	NLS_INITCAP	SOUNDEX	
Character Functions returning number values	All of the functions listed below return number values.		
	ASCII	INSTRB	LENGTHB
	INSTR	LENGTH	

Table 4–1 SQL Function Groups

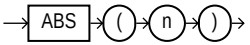
Group	Functions		Description
Date Functions	Date functions operate on values of the DATE datatype. All date functions return a value of DATE datatype, except the MONTHS_BETWEEN function, which returns a number.		
	ADD_MONTHS	NEW_TIME	SYSDATE
	LAST_DAY	NEXT_DAY	TRUNC (Date Function)
	MONTHS_BETWEEN	ROUND (Date Function)	
Conversion Functions	Conversion functions convert a value from one datatype to another. Generally, the form of the function names follows the convention *datatype* TO *datatype*. The first datatype is the input datatype. The second datatype is the output datatype. This section lists the SQL conversion functions.		
	CHARTOROWID	TO_CHAR (date conversion)	TO_LOB
	CONVERT		TO_MULTI_BYTE
	HEXTORAW	TO_CHAR (number conversion)	TO_NUMBER
	RAWTOHEX	TO_DATE	TO_SINGLE_BYTE
	ROWIDTOCHAR		TRANSLATE ... USING
Miscellaneous Single Row Functions	The following single-row functions do not fall into any of the other single-row function categories.		
	BFILENAME	NLS_CHARSET_DECL_LEN	SYS_GUID
	DUMP	NLS_CHARSET_ID	UID
	EMPTY_[B | C]LOB	NLS_CHARSET_NAME	USER
	GREATEST	NVL	USERENV
	LEAST	SYS_CONTEXT	VSIZE
Object Reference Functions	Object functions manipulate REFs, which are references to objects of specified object types. For more information about REFs, see *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals*.		
	DEREF	REF	VALUE
	MAKE_REF	REFTOHEX	

Table 4–1 SQL Function Groups

Group	Functions	Description
Aggregate Functions	Aggregate functions return a single row based on groups of rows, rather than on single rows. Aggregate functions can appear in select lists and HAVING clauses. If you use the GROUP BY clause in a SELECT statement, Oracle divides the rows of a queried table or view into groups. In a query containing a GROUP BY clause, all elements of the select list must be expressions from the GROUP BY clause, expressions containing aggregate functions, or constants. Oracle applies the aggregate functions in the select list to each group of rows and returns a single result row for each group. If you omit the GROUP BY clause, Oracle applies aggregate functions in the select list to all the rows in the queried table or view. You use aggregate functions in the HAVING clause to eliminate groups from the output based on the results of the aggregate functions, rather than on the values of the individual rows of the queried table or view. For more information on the GROUP BY clause and HAVING clauses, see the "GROUP BY Examples" on page 7-553 and the "HAVING" clause on page 7-550. Many aggregate functions accept these options: ■ DISTINCT causes an aggregate function to consider only distinct values of the argument expression. ■ ALL causes an aggregate function to consider all values, including all duplicates. For example, the DISTINCT average of 1, 1, 1, and 3 is 2; the ALL average is 1.5. If neither option is specified, the default is ALL. All aggregate functions except COUNT(*) and GROUPING ignore nulls. You can use the NVL in the argument to an aggregate function to substitute a value for a null. If a query with an aggregate function returns no rows or only rows with nulls for the argument to the aggregate function, the aggregate function returns null. AVG COUNT GROUPING MAX MIN STDDEV SUM VARIANCE	

ABS

Syntax



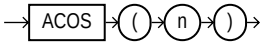
Purpose Returns the absolute value of *n*.

Example

```
SELECT ABS(-15) "Absolute" FROM DUAL;

  Absolute
----------
        15
```

ACOS

Syntax



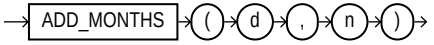
Purpose Returns the arc cosine of n. Inputs are in the range of -1 to 1, and outputs are in the range of 0 to π and are expressed in radians.

Example

```
SELECT ACOS(.3)"Arc_Cosine" FROM DUAL;

Arc_Cosine
----------
1.26610367
```

ADD_MONTHS

Syntax



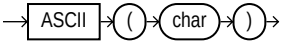
Purpose Returns the date *d* plus *n* months. The argument *n* can be any integer. If *d* is the last day of the month or if the resulting month has fewer days than the day component of *d*, then the result is the last day of the resulting month. Otherwise, the result has the same day component as *d*.

Example

```
SELECT TO_CHAR(
     ADD_MONTHS(hiredate,1),
     'DD-MON-YYYY') "Next month"
     FROM emp
     WHERE ename = 'SMITH';

Next Month
-----------
17-JAN-1981
```

ASCII

Syntax



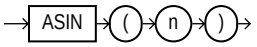
Purpose Returns the decimal representation in the database character set of the first character of *char*. If your database character set is 7-bit ASCII, this function returns an ASCII value. If your database character set is EBCDIC Code Page 500, this function returns an EBCDIC value. There is no similar EBCDIC character function.

Example

```
SELECT ASCII('Q')
     FROM DUAL;

ASCII('Q')
----------
        81
```

ASIN

Syntax



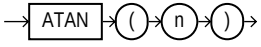
Purpose Returns the arc sine of *n*. Inputs are in the range of -1 to 1, and outputs are in the range of $-\pi/2$ to $\pi/2$ and are expressed in radians.

Example

```
SELECT ASIN(.3) "Arc_Sine" FROM DUAL;

 Arc_Sine
----------
.304692654
```

ATAN

Syntax



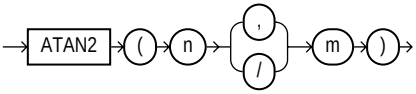
Purpose Returns the arc tangent of *n*. Inputs are in an unbounded range, and outputs are in the range of $-\pi/2$ to $\pi/2$ and are expressed in radians.

Example

```
SELECT ATAN(.3) "Arc_Tangent" FROM DUAL;

Arc_Tangent
----------
.291456794
```

ATAN2

Syntax



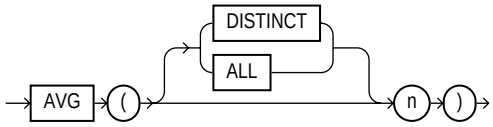
Purpose Returns the arc tangent of *n* and *m*. Inputs are in an unbounded range, and outputs are in the range of -π to π, depending on the signs of *n* and *m*, and are expressed in radians. ATAN2(n,m) is the same as ATAN2(n / m)

Example

```
SELECT ATAN2(.3, .2) "Arc_Tangent2" FROM DUAL;

Arc_Tangent2
------------
  .982793723
```

AVG

Syntax



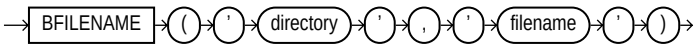
Purpose Returns average value of *n*. See also "Aggregate Functions" on page 4-5.

Example

```
SELECT AVG(sal) "Average"
   FROM emp;

   Average
----------
2077.21429
```

BFILENAME

Syntax



Purpose Returns a BFILE locator that is associated with a physical LOB binary file on the server's file system. A directory is an alias for a full pathname on the server's file system where the files are actually located, and 'filename' is the name of the file in the server's file system.

Neither 'directory' nor 'filename' needs to point to an existing object on the file system at the time you specify BFILENAME. However, you must associate a BFILE value with a physical file before performing subsequent SQL, PL/SQL, DBMS_LOB package, or OCI operations. For more information, see "CREATE DIRECTORY" on page 7-264.

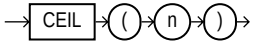
For more information about LOBs, see *Oracle8i Application Developer's Guide - Large Objects (LOBs)* and *Oracle Call Interface Programmer's Guide*.

Example

```
INSERT INTO file_tbl
   VALUES (BFILENAME ('lob_dir1', 'image1.gif'));
```

CEIL

Syntax



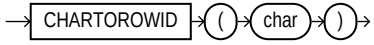
Purpose Returns smallest integer greater than or equal to *n*.

Example

```
SELECT CEIL(15.7) "Ceiling" FROM DUAL;

   Ceiling
----------
        16
```

CHARTOROWID

Syntax



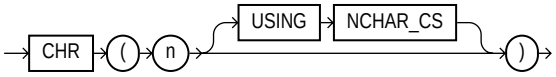
Purpose Converts a value from CHAR or VARCHAR2 datatype to ROWID datatype.

Example

```
SELECT ename FROM emp
   WHERE ROWID = CHARTOROWID('AAAAfZAABAAACp8AAO');

ENAME
----------
LEWIS
```

CHR

Syntax



Purpose Returns the character having the binary equivalent to *n* in either the database character set or the national character set.

If USING NCHAR_CS is *not* specified, this function returns the character having the binary equivalent to *n* as a VARCHAR2 value in the database character set.

If USING NCHAR_CS is specified, this function returns the character having the binary equivalent to *n* as a NVARCHAR2 value in the national character set.

Example 1

```
SELECT CHR(67)||CHR(65)||CHR(84) "Dog"
  FROM DUAL;

Dog
---
CAT
```

Example 2

```
SELECT CHR(16705 USING NCHAR_CS) FROM DUAL;

C
-
A
```

CONCAT

Syntax

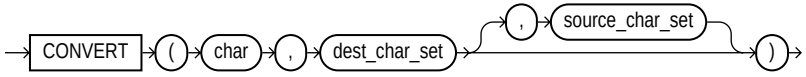
Purpose Returns *char1* concatenated with *char2*. This function is equivalent to the concatenation operator (||). For information on this operator, see "Concatenation Operator" on page 3-3.

Example This example uses nesting to concatenate three character strings:

```
SELECT CONCAT(CONCAT(ename, ' is a '), job) "Job"
FROM emp
WHERE empno = 7900;

Job
-----------------
JAMES is a CLERK
```

CONVERT

Syntax



Purpose Converts a character string from one character set to another.

The *char* argument is the value to be converted.

The *dest_char_set* argument is the name of the character set to which *char* is converted.

The *source_char_set* argument is the name of the character set in which *char* is stored in the database. The default value is the database character set.

Both the destination and source character set arguments can be either literals or columns containing the name of the character set.

For complete correspondence in character conversion, it is essential that the destination character set contains a representation of all the characters defined in the source character set. Where a character does not exist in the destination character set, a replacement character appears. Replacement characters can be defined as part of a character set definition.

Example

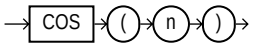
```
SELECT CONVERT('Groß', 'US7ASCII', 'WE8HP')
   "Conversion" FROM DUAL;

Conversion
----------
Gross
```

Common character sets include:

US7ASCII	US 7-bit ASCII character set
WE8DEC	DEC West European 8-bit character set
WE8HP	HP West European Laserjet 8-bit character set
F7DEC	DEC French 7-bit character set
WE8EBCDIC500	IBM West European EBCDIC Code Page 500
WE8PC850	IBM PC Code Page 850
WE8ISO8859P1	ISO 8859-1 West European 8-bit character set

COS

Syntax



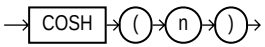
Purpose Returns the cosine of *n* (an angle expressed in radians).

Example

```
SELECT COS(180 * 3.14159265359/180)
"Cosine of 180 degrees" FROM DUAL;

Cosine of 180 degrees
---------------------
                   -1
```

COSH

Syntax



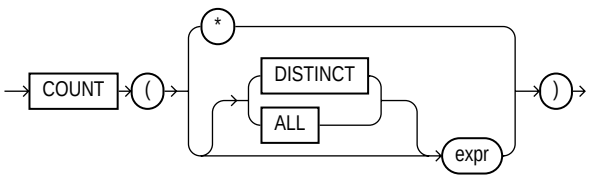
Purpose Returns the hyperbolic cosine of *n*.

Example

```
SELECT COSH(0) "Hyperbolic cosine of 0" FROM DUAL;

Hyperbolic cosine of 0
----------------------
                     1
```

COUNT

Syntax



Purpose Returns the number of rows in the query.

If you specify *expr*, this function returns rows where *expr* is not null. You can count either all rows, or only distinct values of *expr*.

If you specify the asterisk (*), this function returns all rows, including duplicates and nulls. See also "Aggregate Functions" on page 4-5.

Example 1

```
SELECT COUNT(*) "Total"
   FROM emp;

   Total
----------
      18
```

Example 2

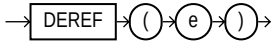
```
SELECT COUNT(job) "Count"
   FROM emp;

   Count
----------
      14
```

Example 3

```
SELECT COUNT(DISTINCT job) "Jobs"
     FROM emp;

       Jobs
----------
        5
```

DEREF

Syntax



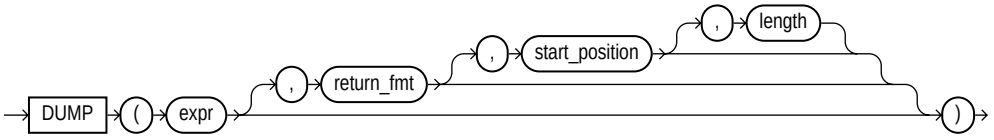
Purpose Returns the object reference of argument *e*. Argument *e* must be an expression that returns a REF to an object.

Example

```
CREATE TYPE emp_type AS OBJECT
   (eno NUMBER, ename VARCHAR2(20), salary NUMBER);
CREATE TABLE emp_table OF emp_type
   (primary key (eno, ename));
CREATE TABLE dept_table
   (dno NUMBER, mgr REF emp_type SCOPE IS emp_table);
INSERT INTO emp_table VALUES (10, 'jack', 50000);
INSERT INTO dept_table SELECT 10, REF(e) FROM emp_table e;
SELECT DEREF(mgr) from dept_table;

DEREF(MGR)(ENO, ENAME, SALARY)
------------------------------------------------------
EMP_TYPE(10, 'jack', 50000)
```

DUMP

Syntax



Purpose Returns a VARCHAR2 value containing the datatype code, length in bytes, and internal representation of *expr*. The returned result is always in the database character set. For the datatype corresponding to each code, see Table 2–1 on page 2-9.

The argument *return_fmt* specifies the format of the return value and can have any of the values listed below.

By default, the return value contains no character set information. To retrieve the character set name of *expr*, specify any of the format values below, plus 1000. For example, a *return_fmt* of 1008 returns the result in octal, plus provides the character set name of *expr*.

8 returns result in octal notation.

10 returns result in decimal notation.

16 returns result in hexadecimal notation.

17 returns result as single characters.

The arguments *start_position* and *length* combine to determine which portion of the internal representation to return. The default is to return the entire internal representation in decimal notation.

If *expr* is null, this function returns 'NULL'.

Example 1

```
SELECT DUMP('abc', 1016)
   FROM DUAL;

DUMP('ABC',1016)
----------------------------------------
Typ=96 Len=3 CharacterSet=WE8DEC: 61,62,63
```

Example 2

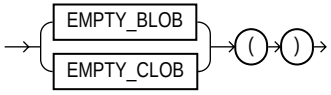
```
SELECT DUMP(ename, 8, 3, 2) "OCTAL"
   FROM emp
   WHERE ename = 'SCOTT';

OCTAL
----------------------------
Type=1 Len=5: 117,124
```

Example 3

```
SELECT DUMP(ename, 10, 3, 2) "ASCII"
   FROM emp
   WHERE ename = 'SCOTT';

ASCII
----------------------------
Type=1 Len=5: 79,84
```

EMPTY_[B | C]LOB

Syntax



Purpose Returns an empty LOB locator that can be used to initialize a LOB variable or in an INSERT or UPDATE statement to initialize a LOB column or attribute to EMPTY. EMPTY means that the LOB is initialized, but not populated with data.

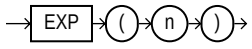
You cannot use the locator returned from this function as a parameter to the DBMS_LOB package or the OCI.

Examples

```
INSERT INTO lob_tab1 VALUES (EMPTY_BLOB());
UPDATE lob_tab1
  SET clob_col = EMPTY_BLOB();
```

EXP

Syntax



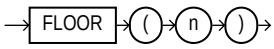
Purpose Returns e raised to the *n*th power, where e = 2.71828183 ...

Example

```
SELECT EXP(4) "e to the 4th power" FROM DUAL;

e to the 4th power
------------------
          54.59815
```

FLOOR

Syntax



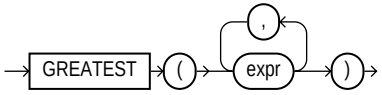
Purpose Returns largest integer equal to or less than *n*.

Example

```
SELECT FLOOR(15.7) "Floor" FROM DUAL;

     Floor
----------
        15
```

GREATEST

Syntax



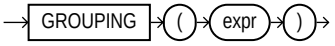
Purpose Returns the greatest of the list of *expr*s. All *expr*s after the first are implicitly converted to the datatype of the first *expr* before the comparison. Oracle compares the *expr*s using nonpadded comparison semantics. Character comparison is based on the value of the character in the database character set. One character is greater than another if it has a higher character set value. If the value returned by this function is character data, its datatype is always VARCHAR2.

Example

```
SELECT GREATEST ('HARRY', 'HARRIOT', 'HAROLD')
   "Greatest" FROM DUAL;

Greatest
--------
HARRY
```

GROUPING

Syntax



Purpose This function is applicable only in a SELECT statement that contains a GROUP BY extension, such as ROLLUP or CUBE. These operations produce superaggregate rows that contain null values representing the set of all values. You can use the GROUPING function to distinguish a null value that represents the set of all values from an actual null value.

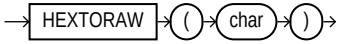
The *expr* in the GROUPING function must match one of the expressions in the GROUP BY clause. The function returns a value of 1 if the value of *expr* in the row is a null representing the set of all values. Otherwise, it returns zero. The datatype of the value returned by the GROUPING function is an Oracle NUMBER datatype. See the *group_by_clause* of the SELECT statement on page 7-549 for a discussion of these terms.

Example

```
SELECT DECODE(GROUPING(dname), 1, 'All Departments',
         dname) AS dname,
   DECODE(GROUPING(job), 1, 'All Jobs', job) AS job,
   COUNT(*) "Total Empl", AVG(sal) * 12 "Average Sal"
   FROM emp, dept
   WHERE dept.deptno = emp.deptno
   GROUP BY ROLLUP (dname, job);

DNAME          JOB       Total Empl Average Sa
-------------- --------- ---------- ----------
ACCOUNTING     CLERK              1      15600
ACCOUNTING     MANAGER            1      29400
ACCOUNTING     PRESIDENT          1      60000
ACCOUNTING     All Jobs           3      35000
RESEARCH       ANALYST            2      36000
RESEARCH       CLERK              2      11400
RESEARCH       MANAGER            1      35700
RESEARCH       All Jobs           5      26100
SALES          CLERK              1      11400
SALES          MANAGER            1      34200
SALES          SALESMAN           4      16800
SALES          All Jobs           6      18800
All Departments All Jobs         14 24878.5714
```

HEXTORAW

Syntax



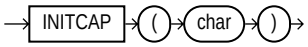
Purpose Converts *char* containing hexadecimal digits to a raw value.

Example

```
INSERT INTO graphics (raw_column)
  SELECT HEXTORAW('7D') FROM DUAL;
```

INITCAP

Syntax



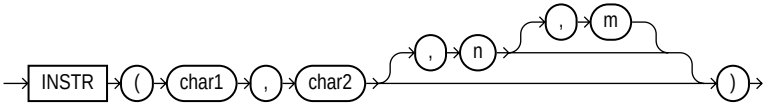
Purpose Returns *char*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

Example

```
SELECT INITCAP('the soap') "Capitals" FROM DUAL;

Capitals
---------
The Soap
```

INSTR

Syntax



Purpose Searches *char1* beginning with its *n*th character for the *m*th occurrence of *char2* and returns the position of the character in *char1* that is the first character of this occurrence. If *n* is negative, Oracle counts and searches backward from the end of *char1*. The value of *m* must be positive. The default values of both *n* and *m* are 1, meaning Oracle begins searching at the first character of *char1* for the first occurrence of *char2*. The return value is relative to the beginning of *char1*, regardless of the value of *n*, and is expressed in characters. If the search is unsuccessful (if *char2* does not appear *m* times after the *n*th character of *char1*) the return value is 0.

Example 1

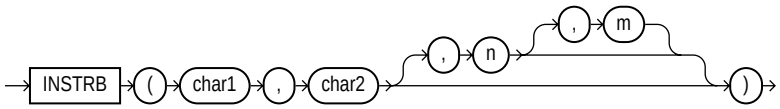
```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2)
  "Instring" FROM DUAL;

  Instring
----------
        14
```

Example 2

```
SELECT INSTR('CORPORATE FLOOR','OR', -3, 2)
"Reversed Instring"
     FROM DUAL;

Reversed Instring
-----------------
                2
```

INSTRB

Syntax



Purpose The same as INSTR, except that *n* and the return value are expressed in bytes, rather than in characters. For a single-byte database character set, INSTRB is equivalent to INSTR.

Example This example assumes a double-byte database character set.

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2)
"Instring in bytes"
FROM DUAL;

Instring in bytes
-----------------
               27
```

LAST_DAY

Syntax

Purpose Returns the date of the last day of the month that contains *d*. You might use this function to determine how many days are left in the current month.

Example 1

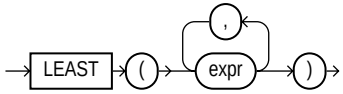
```
SELECT SYSDATE,
   LAST_DAY(SYSDATE) "Last",
   LAST_DAY(SYSDATE) - SYSDATE "Days Left"
   FROM DUAL;

SYSDATE   Last       Days Left
--------- --------- ----------
23-OCT-97 31-OCT-97          8
```

Example 2

```
SELECT TO_CHAR(
  ADD_MONTHS(
    LAST_DAY(hiredate),5),
     'DD-MON-YYYY') "Five months"
  FROM emp
  WHERE ename = 'MARTIN';

Five months
-----------
28-FEB-1982
```

LEAST

Syntax



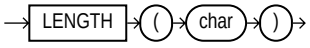
Purpose Returns the least of the list of *expr*s. All *expr*s after the first are implicitly converted to the datatype of the first *expr* before the comparison. Oracle compares the *expr*s using nonpadded comparison semantics. If the value returned by this function is character data, its datatype is always VARCHAR2.

Example

```
SELECT LEAST('HARRY','HARRIOT','HAROLD') "LEAST"
     FROM DUAL;

LEAST
------
HAROLD
```

LENGTH

Syntax



Purpose Returns the length of *char* in characters. If *char* has datatype CHAR, the length includes all trailing blanks. If *char* is null, this function returns null.

Example

```
SELECT LENGTH('CANDIDE') "Length in characters"
FROM DUAL;

Length in characters
--------------------
                   7
```

LENGTHB

Syntax

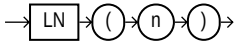
Purpose Returns the length of *char* in bytes. If *char* is null, this function returns null. For a single-byte database character set, LENGTHB is equivalent to LENGTH.

Example This example assumes a double-byte database character set.

```
SELECT LENGTHB ('CANDIDE') "Length in bytes"
     FROM DUAL;

Length in bytes
---------------
             14
```

LN

Syntax



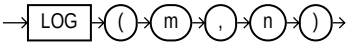
Purpose Returns the natural logarithm of *n*, where *n* is greater than 0.

Example

```
SELECT LN(95) "Natural log of 95" FROM DUAL;

Natural log of 95
-----------------
       4.55387689
```

LOG

Syntax



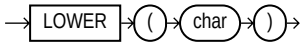
Purpose Returns the logarithm, base *m*, of *n*. The base *m* can be any positive number other than 0 or 1 and *n* can be any positive number.

Example

```
SELECT LOG(10,100) "Log base 10 of 100" FROM DUAL;

Log base 10 of 100
------------------
                 2
```

LOWER

Syntax



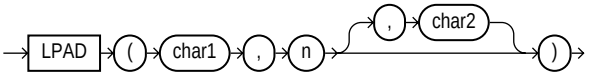
Purpose Returns *char*, with all letters lowercase. The return value has the same datatype as the argument *char* (CHAR or VARCHAR2).

Example

```
SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase"
   FROM DUAL;

Lowercase
--------------------
mr. scott mcmillan
```

LPAD

Syntax



Purpose Returns *char1*, left-padded to length *n* with the sequence of characters in *char2*; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* that fits in *n*.

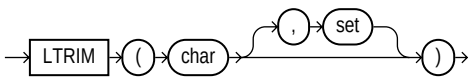
The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Example

```
SELECT LPAD('Page 1',15,'*.') "LPAD example"
     FROM DUAL;

LPAD example
---------------
*.*.*.*.*Page 1
```

LTRIM

Syntax



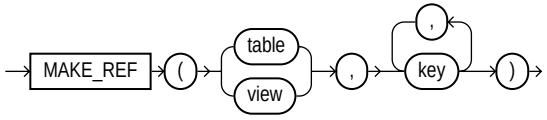
Purpose Removes characters from the left of *char*, with all the leftmost characters that appear in *set* removed; *set* defaults to a single blank. If *char* is a character literal, you must enclose it in single quotes. Oracle begins scanning *char* from its first character and removes all characters that appear in set until reaching a character not in set and then returns the result.

Example

```
SELECT LTRIM('xyxXxyLAST WORD','xy') "LTRIM example"
     FROM DUAL;

LTRIM example
------------
XxyLAST WORD
```

MAKE_REF

Syntax



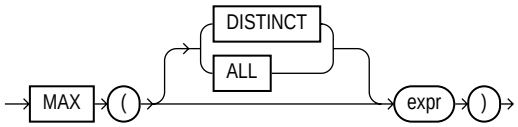
Purpose Creates a REF to a row of an object view or a row in an object table whose object identifier is primary key based. For more information about object views, see *Oracle8i Application Developer's Guide - Fundamentals*.

Example

```
CREATE TABLE emp (eno NUMBER, ename VARCHAR2(20),
   salary NUMBER, PRIMARY KEY (eno, ename));
CREATE TYPE emp_type AS OBJECT
   (eno NUMBER, ename CHAR(20), salary NUMBER);
CREATE VIEW emp_view OF emp_type
   WITH OBJECT IDENTIFIER (eno, ename)
   AS SELECT * FROM emp;
SELECT MAKE_REF(emp_view, 1, 'jack') FROM DUAL;

MAKE_REF(EMP_VIEW,1,'JACK')
----------------------------------------------------------
000067030A0063420D06E06F3C00C1E03400400B40DCB10000001C2
6010001000200290000000000F06008101001401000002A0007000A
8401FE0000001F02C102146A61636B2020202020202020202020202
0202020000000000000000000000000000000000000000000000000
```

MAX

Syntax



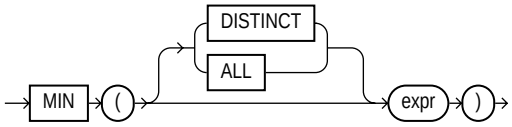
Purpose Returns maximum value of *expr*. See also "Aggregate Functions" on page 4-5.

Example

```
SELECT MAX(sal) "Maximum" FROM emp;

   Maximum
----------
      5000
```

MIN

Syntax



Purpose Returns minimum value of *expr*. See also "Aggregate Functions" on page 4-5.

Example

```
SELECT MIN(hiredate) "Earliest" FROM emp;

Earliest
---------
17-DEC-80
```

MOD

Syntax

Purpose Returns remainder of *m* divided by *n*. Returns *m* if *n* is 0.

Example

```
SELECT MOD(11,4) "Modulus" FROM DUAL;

   Modulus
----------
         3
```

This function behaves differently from the classical mathematical modulus function when *m* is negative. The classical modulus can be expressed using the MOD function with this formula:

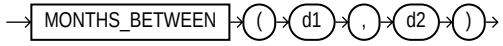
```
m - n * FLOOR(m/n)
```

The following statement illustrates the difference between the MOD function and the classical modulus:

```
SELECT m, n, MOD(m, n),
m - n * FLOOR(m/n) "Classical Modulus"
  FROM test_mod_table;

         M          N   MOD(M,N) Classical Modulus
---------- ---------- ---------- -----------------
        11          4          3                 3
        11         -4          3                -1
       -11          4         -3                 1
       -11         -4         -3                -3
```

MONTHS_BETWEEN

Syntax



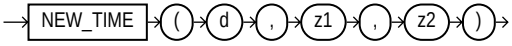
Purpose Returns number of months between dates *d1* and *d2*. If *d1* is later than *d2*, result is positive; if earlier, negative. If *d1* and *d2* are either the same days of the month or both last days of months, the result is always an integer. Otherwise Oracle calculates the fractional portion of the result based on a 31-day month and considers the difference in time components of *d1* and *d2*.

Example

```
SELECT MONTHS_BETWEEN
   (TO_DATE('02-02-1995','MM-DD-YYYY'),
    TO_DATE('01-01-1995','MM-DD-YYYY') ) "Months"
    FROM DUAL;

    Months
----------
1.03225806
```

NEW_TIME

Syntax

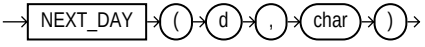


Purpose Returns the date and time in time zone *z2* when date and time in time zone *z1* are *d*. The arguments *z1* and *z2* can be any of these text strings:

AST	Atlantic Standard or Daylight Time
ADT	
BST	Bering Standard or Daylight Time
BDT	
CST	Central Standard or Daylight Time
CDT	
EST	Eastern Standard or Daylight Time
EDT	
GMT	Greenwich Mean Time
HST	Alaska-Hawaii Standard Time or Daylight Time.
HDT	
MST	Mountain Standard or Daylight Time
MDT	
NST	Newfoundland Standard Time
PST	Pacific Standard or Daylight Time
PDT	
YST	Yukon Standard or Daylight Time
YDT	

NEXT_DAY

Syntax



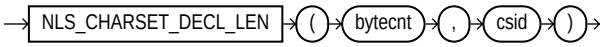
Purpose Returns the date of the first weekday named by *char* that is later than the date *d*. The argument *char* must be a day of the week in your session's date language, either the full name or the abbreviation. The minimum number of letters required is the number of letters in the abbreviated version. Any characters immediately following the valid abbreviation are ignored. The return value has the same hours, minutes, and seconds component as the argument *d*.

Example This example returns the date of the next Tuesday after March 15, 1998.

```
SELECT NEXT_DAY('15-MAR-98','TUESDAY') "NEXT DAY"
     FROM DUAL;

NEXT DAY
---------
16-MAR-98
```

NLS_CHARSET_DECL_LEN

Syntax



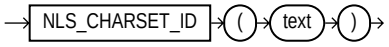
Purpose Returns the declaration width (in number of characters) of an NCHAR column. The *bytecnt* argument is the width of the column. The *csid* argument is the character set ID of the column.

Example

```
SELECT NLS_CHARSET_DECL_LEN
  (200, nls_charset_id('ja16eucfixed'))
   FROM DUAL;

NLS_CHARSET_DECL_LEN(200,NLS_CHARSET_ID('JA16EUCFIXED'))
-------------------------------------------------------
                    100
```

NLS_CHARSET_ID

Syntax



Purpose Returns the NLS character set ID number corresponding to NLS character set name, *text*. The *text* argument is a run-time VARCHAR2 value. The *text* value 'CHAR_CS' returns the server's database character set ID number. The *text* value 'NCHAR_CS' returns the server's national character set ID number.

Invalid character set names return null.

For a list of character set names, see *Oracle8i Reference.*

Example 1

```
SELECT NLS_CHARSET_ID('ja16euc')
  FROM DUAL;

NLS_CHARSET_ID('JA16EUC')
-------------------------
                      830
```

Example 2

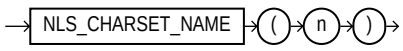
```
SELECT NLS_CHARSET_ID('char_cs')
  FROM DUAL;

 NLS_CHARSET_ID('CHAR_CS')
 -------------------------
                         2
```

Example 3

```
SELECT NLS_CHARSET_ID('nchar_cs')
  FROM DUAL;

NLS_CHARSET_ID('NCHAR_CS')
--------------------------
                         2
```

NLS_CHARSET_NAME

Syntax



Purpose Returns the name of the NLS character set corresponding to ID number *n*. The character set name is returned as a VARCHAR2 value in the database character set.

If *n* is not recognized as a valid character set ID, this function returns null.

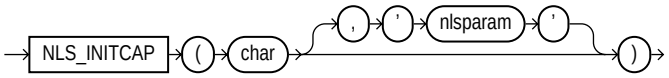
For a list of character set IDs, see *Oracle8i Reference*.

Example

```
SELECT NLS_CHARSET_NAME(2)
  FROM DUAL;

NLS_CH
------
WE8DEC
```

NLS_INITCAP

Syntax



Purpose Returns *char*, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric. The value of '*nlsparam*' can have this form:

```
'NLS_SORT = sort'
```

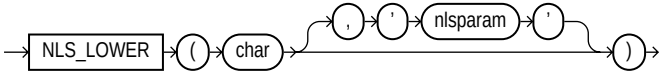
where *sort* is either a linguistic sort sequence or BINARY. The linguistic sort sequence handles special linguistic requirements for case conversions. Note that these requirements can result in a return value of a different length than the *char*. If you omit '*nlsparam*', this function uses the default sort sequence for your session. For information on sort sequences, see *Oracle8i Reference*.

Example

```
SELECT NLS_INITCAP
   ('ijsland', 'NLS_SORT = XDutch') "Capitalized"
   FROM DUAL;

Capital
-------
IJsland
```

NLS_LOWER

Syntax



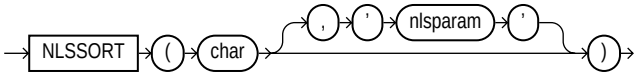
Purpose Returns *char*, with all letters lowercase. The '*nlsparam*' can have the same form and serve the same purpose as in the NLS_INITCAP function.

Example

```
SELECT NLS_LOWER
   ('CITTA''', 'NLS_SORT = XGerman') "Lowercase"
   FROM DUAL;

Lower
-----
cittá
```

NLSSORT

Syntax



Purpose Returns the string of bytes used to sort *char*. The value of '*nlsparams*' can have the form

```
'NLS_SORT = sort'
```

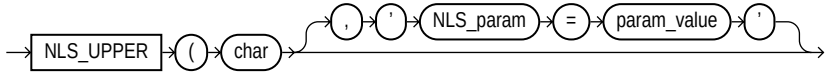
where *sort* is a linguistic sort sequence or BINARY. If you omit '*nlsparams*', this function uses the default sort sequence for your session. If you specify BINARY, this function returns *char*. For information on sort sequences, see *Oracle8i National Language Support Guide*.

Example This function can be used to specify comparisons based on a linguistic sort sequence rather on the binary value of a string:

```
SELECT ename FROM emp
   WHERE NLSSORT (ename, 'NLS_SORT = German')
   > NLSSORT ('S', 'NLS_SORT = German') ORDER BY ename;

ENAME
----------
SCOTT
SMITH
TURNER
WARD
```

NLS_UPPER

Syntax



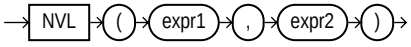
Purpose Returns *char*, with all letters uppercase. The '*nlsparam*' can have the same form and serve the same purpose as in the NLS_INITCAP function.

Example

```
SELECT NLS_UPPER
   ('große', 'NLS_SORT = XGerman') "Uppercase"
     FROM DUAL;

Upper
-----
GROSS
```

NVL

Syntax



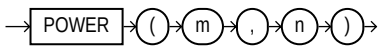
Purpose If *expr1* is null, returns *expr2*; if *expr1* is not null, returns *expr1*. The arguments *expr1* and *expr2* can have any datatype. If their datatypes are different, Oracle converts *expr2* to the datatype of *expr1* before comparing them. The datatype of the return value is always the same as the datatype of *expr1*, unless *expr1* is character data, in which case the return value's datatype is VARCHAR2.

Example

```
SELECT ename, NVL(TO_CHAR(COMM), 'NOT APPLICABLE')
   "COMMISSION" FROM emp
   WHERE deptno = 30;

ENAME      COMMISSION
---------- ------------------------------------
ALLEN      300
WARD       500
MARTIN     1400
BLAKE      NOT APPLICABLE
TURNER     0
JAMES      NOT APPLICABLE
```

POWER

Syntax



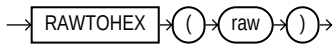
Purpose Returns *m* raised to the *n*th power. The base *m* and the exponent *n* can be any numbers, but if *m* is negative, *n* must be an integer.

Example

```
SELECT POWER(3,2) "Raised" FROM DUAL;

    Raised
----------
         9
```

RAWTOHEX

Syntax



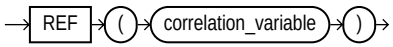
Purpose Converts *raw* to a character value containing its hexadecimal equivalent.

Example

```
SELECT RAWTOHEX(raw_column) "Graphics"
   FROM graphics;

Graphics
--------
7D
```

REF

Syntax



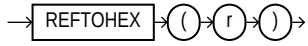
Purpose In a SQL statement, REF takes as its argument a correlation variable (table alias) associated with a row of an object table or an object view. A REF value is returned for the object instance that is bound to the variable or row. For more information about REFs, see *Oracle8i Concepts*.

Examples

```
CREATE TYPE emp_type AS OBJECT
   (eno NUMBER, ename VARCHAR2(20), salary NUMBER);
CREATE TABLE emp_table OF emp_type
   (primary key (eno, ename));
INSERT INTO emp_table VALUES (10, 'jack', 50000);
SELECT REF(e) FROM emp_table e;

REF(E)
----------------------------------------------------
0000280209420D2FEABD9400C3E03400400B40DCB1420D2FEABD930
0C3E03400400B40DCB1004049EE0000
```

REFTOHEX

Syntax



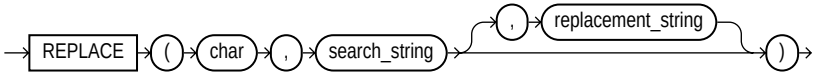
Purpose Converts argument *r* to a character value containing its hexadecimal equivalent.

Example

```
CREATE TYPE emp_type AS OBJECT
   (eno NUMBER, ename VARCHAR2(20), salary NUMBER);
CREATE TABLE emp_table OF emp_type
   (primary key (eno, ename));
CREATE TABLE dept
   (dno NUMBER, mgr REF emp_type SCOPE IS emp);
INSERT INTO emp_table VALUES (10, 'jack', 50000);
INSERT INTO dept SELECT 10, REF(e) FROM emp_table e;
SELECT REFTOHEX(mgr) FROM dept;

REFTOHEX(MGR)
--------------------------------------------------------
0000220208420D2FEABD9400C3E03400400B40DCB1420D2FEABD930
0C3E03400400B40DCB1
```

REPLACE

Syntax



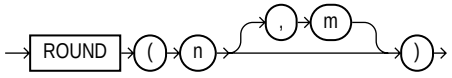
Purpose Returns *char* with every occurrence of *search_string* replaced with *replacement_string*. If *replacement_string* is omitted or null, all occurrences of *search_string* are removed. If *search_string* is null, *char* is returned. This function provides a superset of the functionality provided by the TRANSLATE function. TRANSLATE provides single-character, one-to-one substitution. REPLACE lets you substitute one string for another as well as to remove character strings.

Example

```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
     FROM DUAL;

Changes
--------------
BLACK and BLUE
```

ROUND (Number Function)

Syntax



Purpose Returns *n* rounded to *m* places right of the decimal point. If *m* is omitted, *n* is rounded to 0 places. *m* can be negative to round off digits left of the decimal point. *m* must be an integer.

Example 1

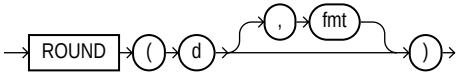
```
SELECT ROUND(15.193,1) "Round" FROM DUAL;

     Round
----------
      15.2
```

Example 2

```
SELECT ROUND(15.193,-1) "Round" FROM DUAL;

     Round
----------
        20
```

ROUND (Date Function)

Syntax



Purpose Returns *d* rounded to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is rounded to the nearest day. See "ROUND and TRUNC Date Functions" on page 4-55 for the permitted format models to use in *fmt*.

Example

```
SELECT ROUND (TO_DATE ('27-OCT-92'),'YEAR')
   "New Year" FROM DUAL;

New Year
---------
01-JAN-93
```

ROWIDTOCHAR

Syntax

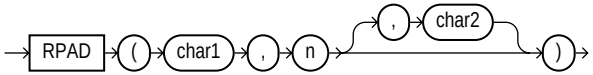
Purpose Converts a rowid value to VARCHAR2 datatype. The result of this conversion is always 18 characters long.

Example

```
SELECT ROWID
     FROM offices
     WHERE
     ROWIDTOCHAR(ROWID) LIKE '%Br1AAB%';

ROWID
------------------
AAAAZ6AABAAABr1AAB
```

RPAD

Syntax



Purpose

Returns *char1*, right-padded to length *n* with *char2*, replicated as many times as necessary; *char2* defaults to a single blank. If *char1* is longer than *n*, this function returns the portion of *char1* that fits in *n*.

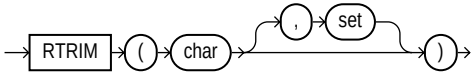
The argument *n* is the total length of the return value as it is displayed on your terminal screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Example

```
SELECT RPAD('MORRISON',12,'ab') "RPAD example"
     FROM DUAL;

RPAD example
-----------------
MORRISONabab
```

RTRIM

Syntax



Purpose

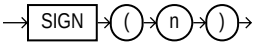
Returns *char*, with all the rightmost characters that appear in *set* removed; *set* defaults to a single blank. If *char* is a character literal, you must enclose it in single quotes. RTRIM works similarly to LTRIM. See "LTRIM" on page 4-22.

Example

```
SELECT RTRIM('BROWNINGyxXxy','xy') "RTRIM e.g."
     FROM DUAL;

RTRIM e.g
-------------
BROWNINGyxX
```

SIGN

Syntax



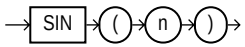
Purpose If *n*<0, the function returns -1. If *n*=0, the function returns 0. If *n*>0, the function returns 1.

Example

```
SELECT SIGN(-15) "Sign" FROM DUAL;

      Sign
----------
        -1
```

SIN

Syntax



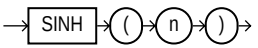
Purpose Returns the sine of *n* (an angle expressed in radians).

Example

```
SELECT SIN(30 * 3.14159265359/180)
 "Sine of 30 degrees" FROM DUAL;

Sine of 30 degrees
------------------
                .5
```

SINH

Syntax



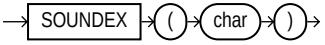
Purpose Returns the hyperbolic sine of *n*.

Example

```
SELECT SINH(1) "Hyperbolic sine of 1" FROM DUAL;

Hyperbolic sine of 1
--------------------
          1.17520119
```

SOUNDEX

Syntax



Purpose Returns a character string containing the phonetic representation of *char*. This function allows you to compare words that are spelled differently, but sound alike in English.

The phonetic representation is defined in *The Art of Computer Programming*, Volume 3: Sorting and Searching, by Donald E. Knuth, as follows:

- Retain the first letter of the string and remove all other occurrences of the following letters: a, e, h, i, o, u, w, y.
- Assign numbers to the remaining letters (after the first) as follows:

  ```
  b, f, p, v = 1
  c, g, j, k, q, s, x, z = 2
  d, t = 3
  l = 4
  m, n = 5
  r = 6
  ```

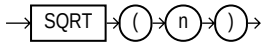
- If two or more letters with the same assigned number are adjacent, remove all but the first.
- Return the first four bytes padded with 0.

Example

```
SELECT ename
     FROM emp
     WHERE SOUNDEX(ename)
         = SOUNDEX('SMYTHE');

ENAME
----------
SMITH
```

SQRT

Syntax



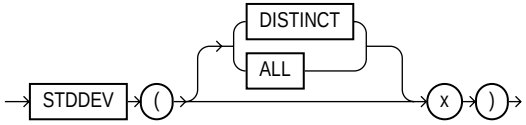
Purpose Returns square root of *n*. The value *n* cannot be negative. SQRT returns a "real" result.

Example

```
SELECT SQRT(26) "Square root" FROM DUAL;

Square root
-----------
5.09901951
```

STDDEV

Syntax



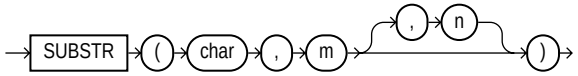
Purpose Returns standard deviation of *x*, a number. Oracle calculates the standard deviation as the square root of the variance defined for the VARIANCE aggregate function. See also "Aggregate Functions" on page 4-5.

Example

```
SELECT STDDEV(sal) "Deviation"
     FROM emp;

 Deviation
----------
1182.50322
```

SUBSTR

Syntax



Purpose Returns a portion of *char*, beginning at character *m*, *n* characters long.

- If *m* is 0, it is treated as 1.
- If *m* is positive, Oracle counts from the beginning of *char* to find the first character.
- If *m* is negative, Oracle counts backwards from the end of *char*.
- If *n* is omitted, Oracle returns all characters to the end of *char*. If n is less than 1, a null is returned.

Floating-point numbers passed as arguments to SUBSTR are automatically converted to integers.

Example 1

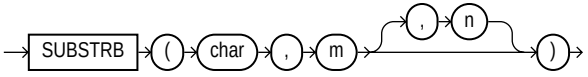
```
SELECT SUBSTR('ABCDEFG',3,4) "Substring"
     FROM DUAL;

Substring
---------
CDEF
```

Example 2

```
SELECT SUBSTR('ABCDEFG',-5,4) "Substring"
     FROM DUAL;

Substring
---------
CDEF
```

SUBSTRB

Syntax



Purpose The same as SUBSTR, except that the arguments *m* and *n* are expressed in bytes, rather than in characters. For a single-byte database character set, SUBSTRB is equivalent to SUBSTR.

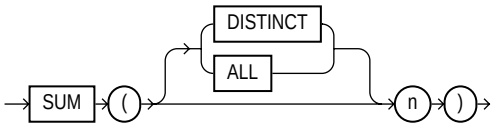
Floating-point numbers passed as arguments to SUBSTRB are automatically converted to integers.

Example Assume a double-byte database character set:

```
SELECT SUBSTRB('ABCDEFG',5,4.2)
     "Substring with bytes"
     FROM DUAL;

Substring with bytes
--------------------
CD
```

SUM

Syntax



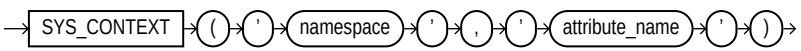
Purpose Returns sum of values of *n*. See also "Aggregate Functions" on page 4-5.

Example

```
SELECT SUM(sal) "Total"
     FROM emp;

     Total
----------
     29081
```

SYS_CONTEXT

Syntax



Purpose Returns the value of *attribute_name* as defined in the package currently associated with the context *namespace*. See "CREATE CONTEXT" on page 7-243. The argument *attribute_name* can have any of the following predefined values:

`'NLS_TERRITORY'`	returns the territory
`'NLS_CURRENCY'`	returns the currency symbol
`'NLS_CALENDAR'`	returns the NLS calendar used for dates
`'NLS_DATE_FORMAT'`	returns the current date format
`'NLS_DATE_LANGUAGE'`	returns the language used for days of the week, months, and so forth, in dates

'NLS_SORT'	indicates whether the sort base is binary or linguistic
'SESSION_USER'	returns the name of the user who logged on
'CURRENT_USER'	returns the current session user name, which may be different from SESSION_USER from within a stored procedure (such as an invoker-rights procedure).
'CURRENT_SCHEMA'	returns the current schema name, which may be changed with an ALTER SESSION SET SCHEMA statement.
'CURRENT SCHEMAID'	returns the current schema ID
'SESSION_USERID	returns the logged on user ID
'CURRENT_USERID'	returns the current session user ID

You can also specify SYS_CONTEXT ('USERENV','IP_ADDRESS') to obtain the IP address of the client if the client is connected to Oracle using the TCP protocol.

Example The following example returns the group number specified as the value for the attribute GROUP_NO in the PL/SQL package that was associated with the context ABC when ABC was created:

```
SELECT SYS_CONTEXT ('abc', 'group_no') "User Group"
   FROM DUAL;

User Group
----------
Sales
```

SYS_GUID

Syntax

Purpose Generates and returns a globally unique identifier (RAW value) made up of 16 bytes. On most platforms, the generated identifier consists of a host identifier and a process or thread identifier of the process or thread invoking the function, and a nonrepeating value (sequence of bytes) for that process or thread.

Example The second line of this example returns the 32-character hexadecimal representation of the 16-byte raw value of the global unique identifier.

```
INSERT INTO my_table VALUES ('BOB', SYS_GUID());
SELECT SYS_GUID() FROM DUAL;
```

SYSDATE

Syntax

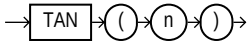
Purpose Returns the current date and time. Requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a CHECK constraint.

Example

```
SELECT TO_CHAR
    (SYSDATE, 'MM-DD-YYYY HH24:MI:SS')"NOW"
     FROM DUAL;

NOW
-------------------
10-29-1993 20:27:11
```

TAN

Syntax



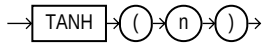
Purpose Returns the tangent of *n* (an angle expressed in radians).

Example

```
SELECT TAN(135 * 3.14159265359/180)
"Tangent of 135 degrees"  FROM DUAL;

Tangent of 135 degrees
----------------------
                    - 1
```

TANH

Syntax



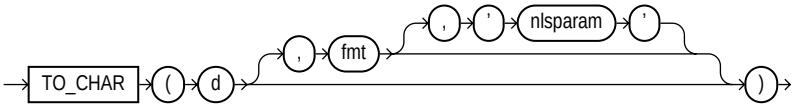
Purpose Returns the hyperbolic tangent of *n*.

Example

```
SELECT TANH(.5) "Hyperbolic tangent of .5"
   FROM DUAL;

Hyperbolic tangent of .5
------------------------
               .462117157
```

TO_CHAR (date conversion)

Syntax



Purpose Converts *d* of DATE datatype to a value of VARCHAR2 datatype in the format specified by the date format *fmt*. If you omit *fmt*, *d* is converted to a VARCHAR2 value in the default date format. For information on date formats, see "Format Models" on page 2-33.

The '*nlsparams*' specifies the language in which month and day names and abbreviations are returned. This argument can have this form:

```
'NLS_DATE_LANGUAGE = language'
```

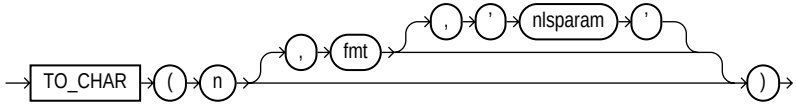
If you omit *nlsparams*, this function uses the default date language for your session.

Example

```
SELECT TO_CHAR(HIREDATE, 'Month DD, YYYY')
    "New date format" FROM emp
    WHERE ename = 'BLAKE';

New date format
------------------
May       01, 1981
```

TO_CHAR (number conversion)

Syntax



Purpose Converts *n* of NUMBER datatype to a value of VARCHAR2 datatype, using the optional number format *fmt*. If you omit *fmt*, *n* is converted to a VARCHAR2 value exactly long enough to hold its significant digits. For information on number formats, see "Format Models" on page 2-33.

The '*nlsparams*' specifies these characters that are returned by number format elements:

- decimal character
- group separator
- local currency symbol
- international currency symbol

This argument can have this form:

```
'NLS_NUMERIC_CHARACTERS = ''dg''
NLS_CURRENCY = ''text''
NLS_ISO_CURRENCY = territory '
```

The characters *d* and *g* represent the decimal character and group separator, respectively. They must be different single-byte characters. Note that within the quoted string, you must use two single quotation marks around the parameter values. Ten characters are available for the currency symbol.

If you omit '*nlsparams*' or any one of the parameters, this function uses the default parameter values for your session.

Example 1 In this example, the output is blank padded to the left of the currency symbol.

```
SELECT TO_CHAR(-10000,'L99G999D99MI') "Amount"
     FROM DUAL;

Amount
---------------
  $10,000.00-
```

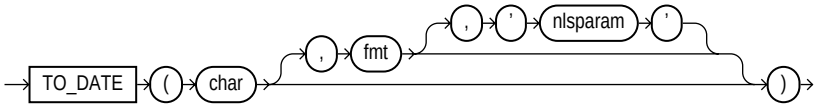
Example 2

```
SELECT TO_CHAR(-10000,'L99G999D99MI',
'NLS_NUMERIC_CHARACTERS = '',.''
NLS_CURRENCY = ''AusDollars'' ') "Amount"
     FROM DUAL;

Amount
-------------------
AusDollars10.000,00-
```

Note: In the optional number format *fmt*, L designates local currency symbol and MI designates a trailing minus sign. See Table 2–7 on page 2-36 for a complete listing of number format elements.

TO_DATE

Syntax



Purpose

Converts *char* of CHAR or VARCHAR2 datatype to a value of DATE datatype. The *fmt* is a date format specifying the format of *char*. If you omit *fmt*, *char* must be in the default date format. If *fmt* is 'J', for Julian, then *char* must be an integer. For information on date formats, see "Format Models" on page 2-33.

The *'nlsparams'* has the same purpose in this function as in the TO_CHAR function for date conversion.

Do not use the TO_DATE function with a DATE value for the *char* argument. The returned DATE value can have a different century value than the original *char*, depending on *fmt* or the default date format.

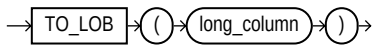
For information on date formats, see "Date Format Models" on page 2-40.

Example

```
INSERT INTO bonus (bonus_date)
  SELECT TO_DATE(
    'January 15, 1989, 11:00 A.M.',
    'Month dd, YYYY, HH:MI A.M.',
     'NLS_DATE_LANGUAGE = American')
     FROM DUAL;
```

TO_LOB

Syntax



Purpose

Converts LONG or LONG RAW values in the column *long_column* to LOB values. You can apply this function only to a LONG or LONG RAW column, and only in the SELECT list of a subquery in an INSERT statement (see "INSERT" on page 7-512).

Before using this function, you must create a LOB column to receive the converted LONG values. To convert LONGs, the LOB column must be of type CLOB or NCLOB. To convert LONG RAWs, the LOB column must be of type BLOB.

Example Given the following tables:

```
CREATE TABLE long_table (n NUMBER, long_col LONG);
CREATE TABLE lob_table (n NUMBER, lob_col CLOB);
```

use this function to convert LONG to LOB values as follows:

```
INSERT INTO lob_table
  SELECT n, TO_LOB(long_col) FROM long_table;
```

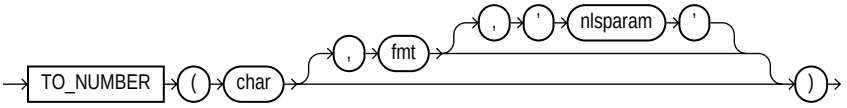
TO_MULTI_BYTE

Syntax

Purpose Returns *char* with all of its single-byte characters converted to their corresponding multibyte characters. Any single-byte characters in *char* that have no multibyte equivalents appear in the output string as single-byte characters. This function is useful only if your database character set contains both single-byte and multibyte characters.

TO_NUMBER

Syntax



Purpose Converts *char*, a value of CHAR or VARCHAR2 datatype containing a number in the format specified by the optional format model *fmt*, to a value of NUMBER datatype.

Example 1

```
UPDATE emp SET sal = sal +
   TO_NUMBER('100.00', '9G999D99')
   WHERE ename = 'BLAKE';
```

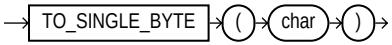
The '*nlsparams*' string in this function has the same purpose as it does in the TO_CHAR function for number conversions. See also "TO_CHAR (number conversion)" on page 4-43.

Example 2

```
SELECT TO_NUMBER('-AusDollars100','L9G999D99',
   ' NLS_NUMERIC_CHARACTERS = '',.''
     NLS_CURRENCY           = ''AusDollars''
   ') "Amount"
     FROM DUAL;

    Amount
----------
      -100
```

TO_SINGLE_BYTE

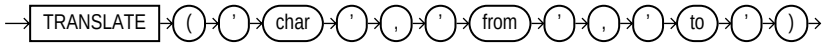
Syntax



Purpose Returns *char* with all of its multibyte character converted to their corresponding single-byte characters. Any multibyte characters in *char* that have no single-byte equivalents appear in the output as multibyte characters. This function is useful only if your database character set contains both single-byte and multibyte characters.

TRANSLATE

Syntax



Purpose Returns *char* with all occurrences of each character in *from* replaced by its corresponding character in *to*. Characters in *char* that are not in *from* are not replaced. The argument *from* can contain more characters than *to*. In this case, the extra characters at the end of *from* have no corresponding characters in *to*. If these extra characters appear in *char*, they are removed from the return value. You cannot use an empty string for *to* to remove all characters in *from* from the return value. Oracle interprets the empty string as null, and if this function has a null argument, it returns null.

Example 1 The following statement translates a license number. All letters 'ABC...Z' are translated to 'X' and all digits '012 . . . 9' are translated to '9':

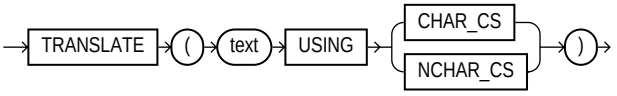
```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'9999999999XXXXXXXXXXXXXXXXXXXXXXXX') "License"
     FROM DUAL;

License
--------
9XXX999
```

Example 2 The following statement returns a license number with the characters removed and the digits remaining:

```
SELECT TRANSLATE('2KRW229',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', '0123456789')
"Translate example"
     FROM DUAL;

Translate example
-----------------
2229
```

TRANSLATE ... USING

Syntax



Purpose Converts *text* into the character set specified for conversions between the database character set and the national character set.

The *text* argument is the expression to be converted.

Specifying the USING CHAR_CS argument converts *text* into the database character set. The output datatype is VARCHAR2.

Specifying the USING NCHAR_CS argument converts *text* into the national character set. The output datatype is NVARCHAR2.

This function is similar to the Oracle CONVERT function, but must be used instead of CONVERT if either the input or the output datatype is being used as NCHAR or NVARCHAR2.

Example 1

```
CREATE TABLE t1 (char_col  CHAR(20),
                 nchar_col nchar(20));
INSERT INTO t1
  VALUES ('Hi', N'Bye');
SELECT * FROM t1;

CHAR_COL     NCHAR_COL
--------     ---------
Hi           Bye
```

Example 2

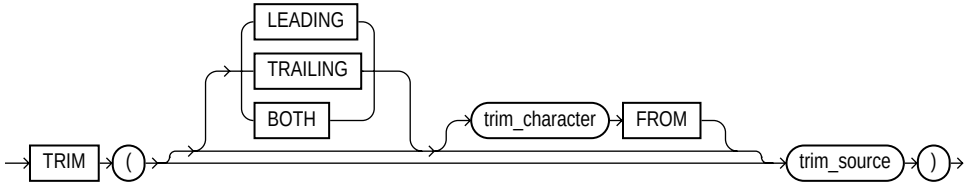
```
UPDATE t1 SET
  nchar_col = TRANSLATE(char_col USING NCHAR_CS);
UPDATE t1 SET
  char_col = TRANSLATE(nchar_col USING CHAR_CS);
SELECT * FROM t1;

CHAR_COL     NCHAR_COL
--------     ---------
Hi           Hi
```

Example 3

```
UPDATE t1 SET
  nchar_col = TRANSLATE('deo' USING NCHAR_CS);
UPDATE t1 SET
  char_col = TRANSLATE(N'deo' USING CHAR_CS);
SELECT * FROM t1;

CHAR_COL     NCHAR_COL
--------     ---------
deo          deo
```

TRIM

Syntax



Purpose enables you to trim heading or trailing characters (or both) from a character string. If *trim_character* or *trim_source* is a character literal, you must enclose it in single quotes.

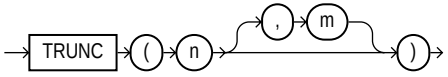
- If you specify LEADING, Oracle removes any leading characters equal to *trim_character*.
- If you specify TRAILING, Oracle removes any trailing characters equal to *trim_character*.
- If you specify BOTH or none of the three, Oracle removes leading and trailing characters equal to *trim_character*.
- If you do not specify *trim_character*, the default value is a blank space.
- The function returns a value with datatype VARCHAR2. The maximum length of the value is the length of *trim_source*.
- If either *trim_source* or *trim_character* is a null value, then the TRIM function returns a null value.

Example This example trims leading and trailing zeroes from a number:

```
SELECT TRIM (0 FROM 0009872348900) "TRIM Example"
   FROM DUAL;

TRIM example
------------
    98723489
```

TRUNC (Number Function)

Syntax



Purpose Returns *n* truncated to *m* decimal places. If *m* is omitted, *n* is truncated to 0 places. *m* can be negative to truncate (make zero) *m* digits left of the decimal point.

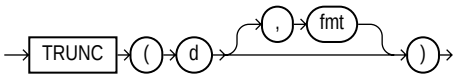
Examples

```
SELECT TRUNC(15.79,1) "Truncate" FROM DUAL;

  Truncate
----------
      15.7
```

```
SELECT TRUNC(15.79,-1) "Truncate" FROM DUAL;

  Truncate
----------
        10
```

TRUNC (Date Function)

Syntax



Purpose Returns *d* with the time portion of the day truncated to the unit specified by the format model *fmt*. If you omit *fmt*, *d* is truncated to the nearest day. See "ROUND and TRUNC Date Functions" on page 4-55 for the permitted format models to use in *fmt*.

Example

```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR')
  "New Year" FROM DUAL;

New Year
---------
01-JAN-92
```

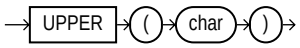
UID

Syntax

Purpose Returns an integer that uniquely identifies the current user.

UPPER

Syntax



Purpose Returns *char*, with all letters uppercase. The return value has the same datatype as the argument *char*.

Example

```
SELECT UPPER('Large') "Uppercase"
     FROM DUAL;

Upper
-----
LARGE
```

USER

Syntax

Purpose Returns the current Oracle user with the datatype VARCHAR2. Oracle compares values of this function with blank-padded comparison semantics.

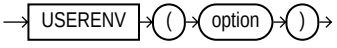
In a distributed SQL statement, the UID and USER functions identify the user on your local database. You cannot use these functions in the condition of a CHECK constraint.

Example

```
SELECT USER, UID FROM DUAL;

USER                                UID
------------------------------ ----------
SCOTT                                19
```

USERENV

Syntax



Purpose Returns information of VARCHAR2 datatype about the current session. This information can be useful for writing an application-specific audit trail table or for determining the language-specific characters currently used by your session. You cannot use USERENV in the condition of a CHECK constraint. The argument *option* can have any of these values:

`'ISDBA'`	returns 'TRUE' if you currently have the ISDBA role enabled and 'FALSE' if you do not.
`'LANGUAGE'`	returns the language and territory currently used by your session along with the database character set in this form: `language_territory.characterset`

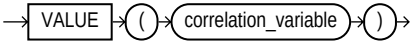
`'TERMINAL'`	returns the operating system identifier for your current session's terminal. In distributed SQL statements, this option returns the identifier for your local session. In a distributed environment, this is supported only for remote SELECTs, not for remote INSERTs, UPDATEs, or DELETEs.
`'SESSIONID'`	returns your auditing session identifier. You cannot use this option in distributed SQL statements.
`'ENTRYID'`	returns available auditing entry identifier. You cannot use this option in distributed SQL statements. To use this keyword in USERENV, the initialization parameter AUDIT_TRAIL must be set to TRUE.
`'LANG'`	Returns the ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.
`'INSTANCE'`	Returns the instance identification number of the current instance.
`'CLIENT_INFO'`	returns up to 64 bytes of user session information that can be stored by an application using the DBMS_ APPLICATION_INFO package. **CAUTION:** Some commercial applications may be using this context value. Check the applicable documentation for those applications to determine what restrictions they may impose on use of this context area. Oracle recommends that you use the application context feature or the SYS_CONTEXT function with the USERENV option. These alternatives are more secure and flexible. For information on application context, see *Oracle8i Concepts*. See also "CREATE CONTEXT" on page 7-243 and "SYS_CONTEXT" on page 4-40.

Example

```
SELECT USERENV('LANGUAGE') "Language" FROM DUAL;

Language
----------------------------------
AMERICAN_AMERICA.WE8DEC
```

VALUE

Syntax



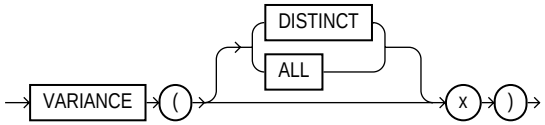
Purpose In a SQL statement, VALUE takes as its argument a correlation variable (table alias) associated with a row of an object table and returns object instances stored in the object table. The type of the object instances is the same type as the object table.

Example

```
CREATE TYPE emp_type AS OBJECT
   (eno NUMBER, ename VARCHAR2(20), salary NUMBER);
CREATE TABLE emp_table OF emp_type
   (primary key (eno, ename));
INSERT INTO emp_table VALUES (10, 'jack', 50000);
SELECT VALUE(e) FROM emp_table e;

VALUE(E)(ENO, ENAME, SALARY)
----------------------------------------------------
EMP_TYPE(10, 'jack', 50000)
```

VARIANCE

Syntax



Purpose Returns variance of *x*, a number. Oracle calculates the variance of *x* using this formula:

$$\frac{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left[\sum_{i=1}^{n} x_i\right]^2}{n-1}$$

where:

x_i is one of the elements of *x*.

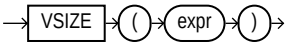
n is the number of elements in the set *x*. If n is 1, the variance is defined to be 0. See also "Aggregate Functions" on page 4-5.

Example

```
SELECT VARIANCE(sal) "Variance"
     FROM emp;

Variance
----------
1389313.87
```

VSIZE

Syntax



Purpose Returns the number of bytes in the internal representation of *expr*. If *expr* is null, this function returns null.

Example

```
SELECT ename, VSIZE (ename) "BYTES"
  FROM emp
  WHERE deptno = 10;

ENAME           BYTES
---------- ----------
CLARK                5
KING                 4
MILLER               6
```

ROUND and TRUNC Date Functions

Table 4–2 lists the format models you can use with the ROUND and TRUNC date functions and the units to which they round and truncate dates. The default model, 'DD', returns the date rounded or truncated to the day with a time of midnight.

Table 4–2 Date Format Models for the ROUND and TRUNC Date Functions

Format Model	Rounding or Truncating Unit
CC SCC	One greater than the first two digits of a four-digit year.
SYYYY YYYY YEAR SYEAR YYY YY Y	Year (rounds up on July 1)
IYYY IY IY I	ISO Year
Q	Quarter (rounds up on the sixteenth day of the second month of the quarter)

Table 4–2 (Cont.) Date Format Models for the ROUND and TRUNC Date Functions

Format Model	Rounding or Truncating Unit
MONTH MON MM RM	Month (rounds up on the sixteenth day)
WW	Same day of the week as the first day of the year.
IW	Same day of the week as the first day of the ISO year.
W	Same day of the week as the first day of the month.
DDD DD J	Day
DAY DY D	Starting day of the week
HH HH12 HH24	Hour
MI	Minute

The starting day of the week used by the format models DAY, DY, and D is specified implicitly by the initialization parameter NLS_TERRITORY. For information on this parameter, see *Oracle8i Reference*.

User-Defined Functions

You can write user functions in PL/SQL or Java to provide functionality that is not available in SQL or SQL functions. User functions can appear in a SQL statement anywhere SQL functions can appear, that is, wherever an expression can occur.

For example, user functions can be used in the following:

- The select list of a SELECT statement
- The condition of a WHERE clause
- CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses
- The VALUES clause of an INSERT statement

- The SET clause of an UPDATE statement

For information on creating functions, including restrictions on user-defined functions, see "CREATE FUNCTION" on page 7-266. For a complete description on the creation and use of user functions, see *Oracle8i Application Developer's Guide - Fundamentals*.

Prerequisites

User functions must be created as top-level functions or declared with a package specification before they can be named within a SQL statement. Create user functions as top-level functions by using the CREATE FUNCTION statement described in "CREATE FUNCTION" on page 7-266. To specify packaged functions, see "CREATE PACKAGE" on page 7-325.

To use a user function in a SQL expression, you must own or have EXECUTE privilege on the user function. To query a view defined with a user function, you must have SELECT privileges on the view. No separate EXECUTE privileges are needed to select from the view.

Name Precedence

Within a SQL statement, the names of database columns take precedence over the names of functions with no parameters. For example, if user SCOTT creates the following two objects in his own schema:

```
CREATE TABLE emp(new_sal NUMBER, ...);
CREATE FUNCTION new_sal RETURN NUMBER IS BEGIN ... END;
```

then in the following two statements, the reference to NEW_SAL refers to the column EMP.NEW_SAL:

```
SELECT new_sal FROM emp;
SELECT emp.new_sal FROM emp;
```

To access the function NEW_SAL, you would enter:

```
SELECT scott.new_sal FROM emp;
```

Here are some sample calls to user functions that are allowed in SQL expressions.

```
circle_area (radius)
payroll.tax_rate (empno)
scott.payroll.tax_rate (dependent, empno)@ny
```

Example For example, to call the TAX_RATE user function from schema SCOTT, execute it against the SS_NO and SAL columns in TAX_TABLE, and place the results in the variable INCOME_TAX, specify the following:

```
SELECT scott.tax_rate (ss_no, sal)
    INTO income_tax
    FROM tax_table
    WHERE ss_no = tax_id;
```

Naming Conventions

If only one of the optional schema or package names is given, the first identifier can be either a schema name or a package name. For example, to determine whether PAYROLL in the reference PAYROLL.TAX_RATE is a schema or package name, Oracle proceeds as follows:

- Check for the PAYROLL package in the current schema.
- If a PAYROLL package is not found, look for a schema name PAYROLL that contains a top-level TAX_RATE function. If no such function is found, return an error.
- If the PAYROLL package is found in the current schema, look for a TAX_RATE function in the PAYROLL package. If no such function is found, return an error.

You can also refer to a stored top-level function using any synonym that you have defined for it.

5

Expressions, Conditions, and Queries

The ideal condition would be, I admit, that men should be right by instinct.

Sophocles, *Oedipus Rex*

This chapter describes how to combine the values, operators, and functions described in earlier chapters evaluate to a value. Topics include:

- Expressions
- Conditions
- Queries and Subqueries

Expressions

An *expression* is a combination of one or more values, operators, and SQL functions that evaluate to a value. An expression generally assumes the datatype of its components.

This simple expression evaluates to 4 and has datatype NUMBER (the same datatype as its components):

```
2*2
```

The following expression is an example of a more complex expression that uses both functions and operators. The expression adds seven days to the current date, removes the time component from the sum, and converts the result to CHAR datatype:

```
TO_CHAR(TRUNC(SYSDATE+7))
```

You can use expressions in:

- The select list of the SELECT statement

- A condition of the WHERE clause and HAVING clause
- The CONNECT BY, START WITH, and ORDER BY clauses
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement

For example, you could use an expression in place of the quoted string 'smith' in this UPDATE statement SET clause:

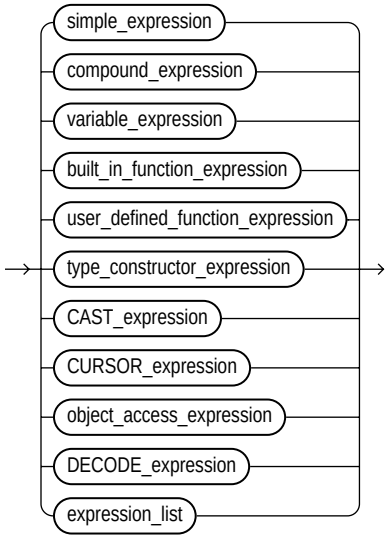
```
SET ename = 'smith';
```

This SET clause has the expression LOWER(ename) instead of the quoted string 'smith':

```
SET ename = LOWER(ename);
```

Expressions have several forms, as shown in the following syntax:

expr::=



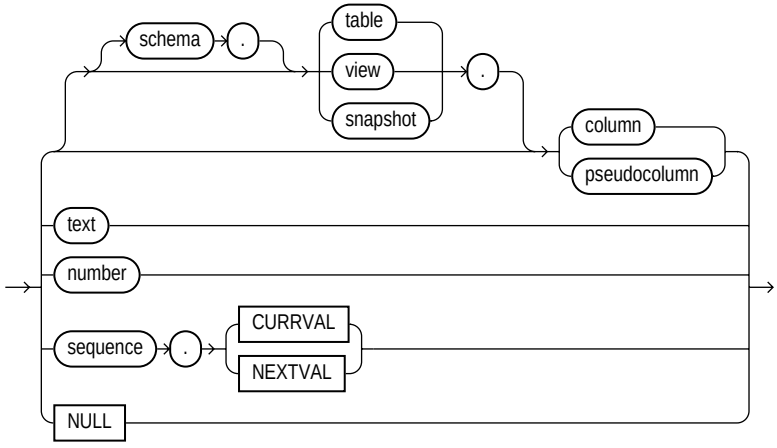
Oracle does not accept all forms of expressions in all parts of all SQL statements. You must use appropriate expression notation whenever *expr* appears in conditions, SQL functions, or SQL statements in other parts of this reference. The description of each statement in Chapter 7, "SQL Statements", documents the

restrictions on the expressions in the statement. The sections that follow describe and provide examples of the various forms of expressions.

Simple Expressions

A simple expression specifies column, pseudocolumn, constant, sequence number, or NULL.

simple_expression::=



In addition to the schema of a user, *schema* can also be "PUBLIC" (double quotation marks required), in which case it must qualify a public synonym for a table, view, or snapshot. Qualifying a public synonym with "PUBLIC" is supported only in data manipulation language (DML) statements, not data definition language (DDL) statements.

The *pseudocolumn* can be either LEVEL, ROWID, or ROWNUM. You can use a pseudocolumn only with a table, not with a view or snapshot. NCHAR and NVARCHAR2 are not valid pseudocolumn datatypes. For more information on pseudocolumns, see "Pseudocolumns" on page 2-51.

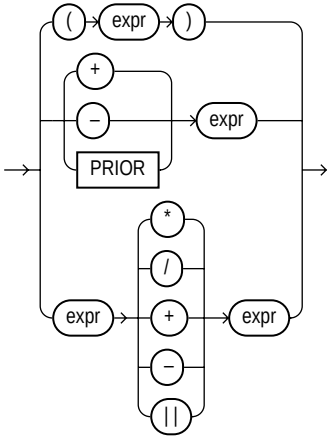
Some valid simple expressions are:

```
emp.ename
'this is a text string'
10
N'this is an NCHAR string'
```

Compound Expressions

A compound expression specifies a combination of other expressions.

compound_expression::=



Note that some combinations of functions are inappropriate and are rejected. For example, the LENGTH function is inappropriate within an aggregate function.

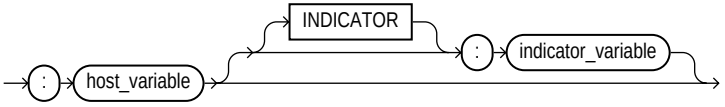
Some valid compound expressions are:

```
('CLARK' || 'SMITH')
LENGTH('MOOSE') * 57
SQRT(144) + 72
my_fun(TO_CHAR(sysdate,'DD-MMM-YY')
```

Variable Expressions

A variable expression specifies a host variable with an optional indicator variable. Note that this form of expression can appear only in embedded SQL statements or SQL statements processed in an Oracle Call Interface (OCI) program.

variable_expression::=



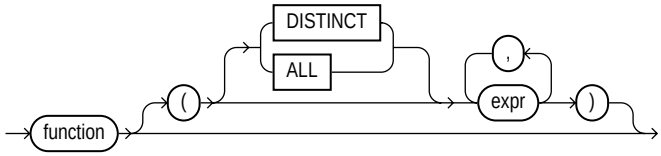
Some valid variable expressions are:

```
:employee_name INDICATOR :employee_name_indicator_var
:department_location
```

Built-In Function Expressions

A built-in function expression specifies a call to a single-row SQL function.

built_in_function_expression::=



Some valid built-in function expressions are:

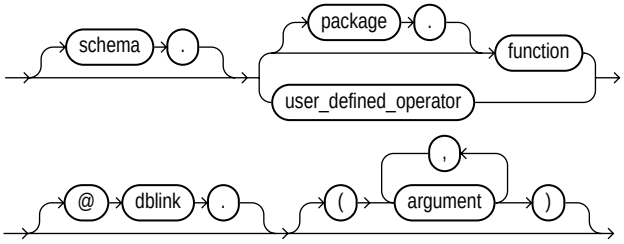
```
LENGTH('BLAKE')
ROUND(1234.567*43)
SYSDATE
```

For information on built-in functions, see "SQL Functions" on page 4-1. See also "Aggregate Functions" on page 4-5.

User-Defined Function Expressions

A user-defined function expression specifies a call to a user-defined function.

user_defined_function_expression::=



Some valid user-defined function expressions are:

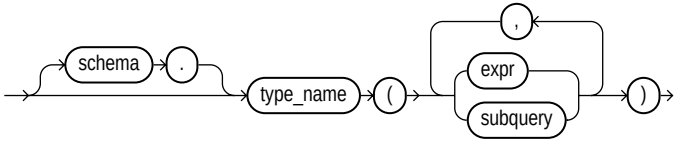
```
circle_area(radius)
payroll.tax_rate(empno)
scott.payrol.tax_rate(dependents, empno)@ny
```

For information on user-defined functions, see "User-Defined Functions" on page 4-56. For information on user-defined operators, see "CREATE OPERATOR" on page 7-320 and *Oracle8i Data Cartridge Developer's Guide.*

Type Constructor Expressions

A type constructor expression specifies a call to a type constructor. The *argument* to the type constructor is any expression or subquery.

type_constructor_expression::=



If *type_name* is an **object type**, then the argument list must be an ordered list, where the first argument is a value whose type matches the first attribute of the object type, the second argument is a value whose type matches the second attribute of the object type, and so on. The total number of arguments to the constructor must match the total number of attributes of the object type.

If *type_name* is a **varray or nested table type**, then the argument list can contain zero or more arguments. Zero arguments implies construction of an empty collection. Otherwise, each argument corresponds to an element value whose type is the element type of the collection type.

If *type_name* is an **object type, a varray, or a nested table type**, the maximum number of arguments it can contain is 1000 minus some overhead.

Expression Example This example shows the use of an expression in the call to a type constructor.

```
CREATE TYPE address_t AS OBJECT
  (no NUMBER, street CHAR(31), city CHAR(21), state CHAR(3), zip NUMBER);
CREATE TYPE address_book_t AS TABLE OF address_t;
DECLARE
  /* Object Type variable initialized via Object Type Constructor */
  myaddr address_t = address_t(500, 'Oracle Parkway', 'Redwood Shores', 'CA',  94065);
```

```
  /* nested table variable initialized to an empty table via a constructor*/
  alladdr address_book_t = address_book_t();
BEGIN
  /* below is an example of a nested table constructor with two elements
     specified, where each element is specified as an object type constructor. */
  insert into employee values (666999, address_book_t(address_t(500,
     'Oracle  Parkway', 'Redwood Shores', 'CA', 94065), address_t(400,
     'Mission Street', 'Fremont', 'CA', 94555)));
END;
```

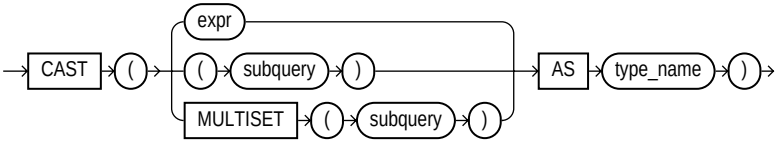
Subquery Example This example illustrates the use of a subquery in the call to the type constructor.

```
CREATE TYPE employee AS OBJECT (
   empno NUMBER,
   ename VARCHAR2(20));
CREATE TABLE emptbl of EMPLOYEE;
INSERT INTO emptbl VALUES(7377, 'JOHN');
CREATE TYPE project AS OBJECT (
   pname VARCHAR2(25),
   empref REF employee);
CREATE TABLE depttbl (dno number, proj project);
INSERT INTO depttbl values(10, project('SQL Extensions',
                                       (SELECT REF(p) FROM emptbl p
                                        WHERE ename='JOHN')));
```

CAST Expressions

A CAST expression converts one built-in datatype or collection-typed value into another built-in datatype or collection-typed value.

CAST_expression::=



CAST allows you to convert built-in datatypes or collection-typed values of one type into another built-in datatype or collection type. You can cast an unnamed operand (such as a date or the result set of a subquery) or a named collection (such as a varray or a nested table) into a type-compatible datatype or named collection.

The *type_name* must be the name of a built-in datatype or collection type and the *operand* must be a built-in datatype or must evaluate to a collection value.

For the operand, *expr* can be either a built-in datatype or a collection type, and *subquery* must return a single value of collection type or built-in type. MULTISET informs Oracle to take the result set of the subquery and return a collection value. Table 5–1 shows which built-in datatypes can be cast into which other built-in datatypes. (CAST does not support LONG, LONG RAW, or any of the LOB datatypes.)

Table 5–1 Casting Built-In Datatypes

From/ To	CHAR, VARCHAR2	NUMBER	DATE	RAW	ROWID, UROWID	NCHAR, NVARCHAR2
CHAR, VARCHAR2	X	X	X	X	X	
NUMBER	X	X				
DATE	X		X			
RAW	X			X		
ROWID, UROWID	X				X[a]	
NCHAR, NVARCHAR2		X	X	X	X	X

[a] You cannot cast a UROWID to a ROWID if the UROWID contains the value of a ROWID of an index-organized table.

To cast a named collection type into another named collection type, the elements of both collections must be of the same type.

If the result set of *subquery* can evaluate to multiple rows, you must specify the MULTISET keyword. The rows resulting from the subquery form the elements of the collection value into which they are cast. Without the MULTISET keyword, the subquery is treated as a scalar subquery, which is not supported in the CAST expression. In other words, scalar subqueries as arguments of the CAST operator are not valid in Oracle8*i*.

Built-In Datatype Examples

```
SELECT CAST ('1997-10-22' AS DATE) FROM DUAL;
SELECT * FROM t1 WHERE CAST (ROWID AS VARCHAR2) = '01234';
```

Collection Examples The CAST examples that follow use the following user-defined types and tables:

```
CREATE TYPE address_t AS OBJECT
      (no NUMBER, street CHAR(31), city CHAR(21), state CHAR(2));
CREATE TYPE address_book_t AS TABLE OF address_t;
CREATE TYPE address_array_t AS VARRAY(3) OF address_t;
CREATE TABLE emp_address (empno NUMBER, no NUMBER, street CHAR(31),
                          city  CHAR(21), state CHAR(2));
CREATE TABLE employees (empno NUMBER, name CHAR(31));
CREATE TABLE dept (dno NUMBER, addresses address_array_t);
```

This example casts a subquery:

```
SELECT e.empno, e.name, CAST(MULTISET(SELECT ea.no, ea.street,
                                             ea.city, ea.state
                                      FROM emp_address ea
                                      WHERE ea.empno = e.empno)
                        AS address_book_t)
  FROM employees e;
```

CAST converts a varray type column into a nested table:

```
SELECT CAST(d.addresses AS address_book_t)
   FROM dept d
   WHERE d.dno = 111);
```

The following example casts a MULTISET expression with an ORDER BY clause:

```
CREATE TABLE projects (empid NUMBER, projname VARCHAR2(10));
CREATE TABLE employees (empid NUMBER, ename VARCHAR2(10));
CREATE TYPE projname_table_type AS TABLE OF VARCHAR2(10);
```

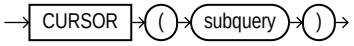
An example of a MULTISET expression with the above schema is:

```
SELECT e.ename, CAST(MULTISET(SELECT p.projname
                              FROM projects p
                              WHERE p.empid=e.empid
                              ORDER BY p.projname)
  AS projname_table_type)
   FROM employees e;
```

CURSOR Expressions

A CURSOR expression returns a nested cursor. This form of expression is similar to the PL/SQL REF cursor.

CURSOR_expression::=



A nested cursor is implicitly opened when the containing row is fetched from the parent cursor. The nested cursor is closed only when:

- The nested cursor is explicitly closed by the user
- The parent cursor is reexecuted
- The parent cursor is closed
- The parent cursor is cancelled
- An error arises during fetch on one of its parent cursors (it is closed as part of the clean-up)

Restrictions: The following restrictions apply to the CURSOR expression:

- Nested cursors can appear only in a SELECT statement that is not nested in any other query expression, except when it is a subquery of the CURSOR expression itself.
- Nested cursors can appear only in the outermost SELECT list of the query specification.
- Nested cursors cannot appear in views.
- You cannot perform BIND and EXECUTE operations on nested cursors.

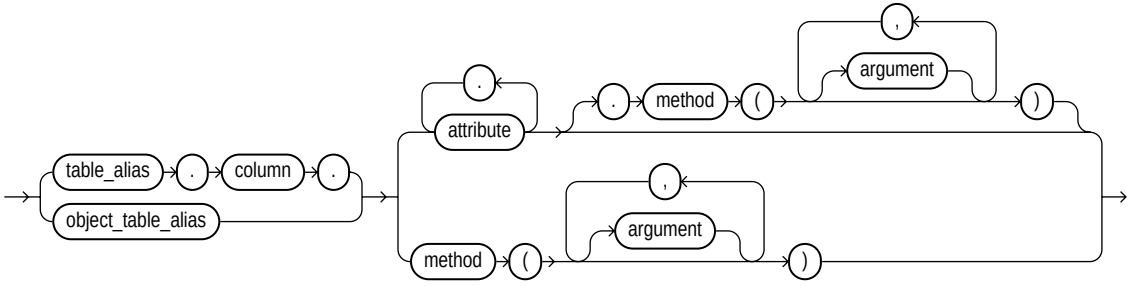
Example

```
SELECT d.deptno, CURSOR(SELECT e.empno, CURSOR(SELECT p.projnum,
                                                      p.projname
                                               FROM   projects p
                                               WHERE  p.empno = e.empno)
                        FROM TABLE(d.employees) e)
  FROM dept d
  WHERE d.dno = 605;
```

Object Access Expressions

An object access expression specifies attribute reference and method invocation.

object_access_expression::=



The `column` parameter can be an object or REF column.

When a type's member function is invoked in the context of a SQL statement, if the SELF argument is null, Oracle returns null and the function is not invoked.

Examples in this section use the following user-defined types and tables:

```
CREATE OR REPLACE TYPE employee_t AS OBJECT
   (empid NUMBER,
    name VARCHAR2(31),
    birthdate DATE,
    MEMBER FUNCTION age RETURN NUMBER,
    PRAGMA RESTRICT_REFERENCES (age, RNPS, WNPS, WNDS)
    );

CREATE OR REPLACE TYPE BODY employee_t AS
    MEMBER FUNCTION age RETURN NUMBER IS
       var NUMBER;
       BEGIN
          var := TRUNC(MONTHS_BETWEEN(SYSDATE, birthdate) /12);
          RETURN(var);
       END;
    END;

CREATE TABLE department (dno NUMBER, manager EMPLOYEE_T);
```

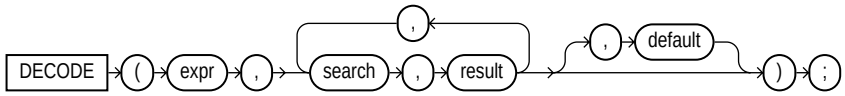
Examples The following examples update and select from the object columns and method defined above.

```
UPDATE department d
  SET d.manager.empid = 100;
SELECT d.manager.name, d.manager.age()
  FROM department d;
```

DECODE Expressions

A DECODE expression uses the special DECODE syntax:

DECODE_expression::=



To evaluate this expression, Oracle compares *expr* to each *search* value one by one. If *expr* is equal to a *search*, Oracle returns the corresponding *result*. If no match is found, Oracle returns *default*, or, if *default* is omitted, returns null. If *expr* and *search* contain character data, Oracle compares them using nonpadded comparison semantics. For information on these semantics, see the section"Datatype Comparison Rules" on page 2-27.

The *search*, *result*, and *default* values can be derived from expressions. Oracle evaluates each *search* value only before comparing it to *expr*, rather than evaluating all *search* values before comparing any of them with *expr*. Consequently, Oracle never evaluates a *search* if a previous *search* is equal to *expr*.

Oracle automatically converts *expr* and each *search* value to the datatype of the first *search* value before comparing. Oracle automatically converts the return value to the same datatype as the first *result*. If the first *result* has the datatype CHAR or if the first *result* is null, then Oracle converts the return value to the datatype VARCHAR2. For information on datatype conversion, see "Data Conversion" on page 2-31.

In a DECODE expression, Oracle considers two nulls to be equivalent. If *expr* is null, Oracle returns the *result* of the first *search* that is also null.

The maximum number of components in the DECODE expression, including *expr*, *search*es, *result*s, and *default* is 255.

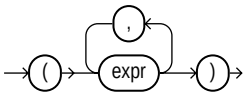
Example This expression decodes the value DEPTNO. If DEPTNO is 10, the expression evaluates to 'ACCOUNTING'; if DEPTNO is 20, it evaluates to 'RESEARCH'; etc. If DEPTNO is not 10, 20, 30, or 40, the expression returns 'NONE'.

```
DECODE (deptno,10, 'ACCOUNTING',
               20, 'RESEARCH',
               30, 'SALES',
               40, 'OPERATION',
                   'NONE')
```

Expression List

An expression list is a series of expressions separated by a comma. The entire series is enclosed in parentheses.

expression_list::=



An expression list can contain up to 1000 expressions. Some valid expression lists are:

```
(10, 20, 40)
('SCOTT', 'BLAKE', 'TAYLOR')
(LENGTH('MOOSE') * 57, -SQRT(144) + 72, 69)
```

Conditions

A condition specifies a combination of one or more expressions and logical operators that evaluates to either TRUE, FALSE, or unknown. You must use this syntax whenever *condition* appears in SQL statements in Chapter 7, "SQL Statements".

You can use a condition in the WHERE clause of these statements:

- DELETE
- SELECT
- UPDATE

You can use a condition in any of these clauses of the SELECT statement:

- WHERE
- START WITH
- CONNECT BY
- HAVING

A condition could be said to be of the "logical" datatype, although Oracle does not formally support such a datatype.

The following simple condition always evaluates to TRUE:

```
1 = 1
```

The following more complex condition adds the SAL value to the COMM value (substituting the value 0 for null) and determines whether the sum is greater than the number constant 2500:

```
NVL(sal, 0) + NVL(comm, 0) > 2500
```

Logical operators can combine multiple conditions into a single condition. For example, you can use the AND operator to combine two conditions:

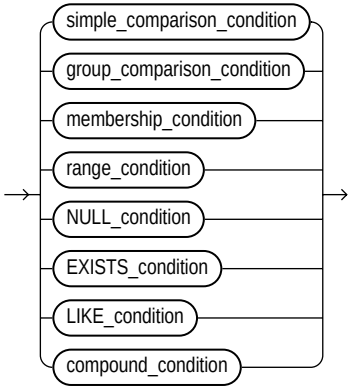
```
(1 = 1) AND (5 < 7)
```

Here are some valid conditions:

```
name = 'SMITH'
emp.deptno = dept.deptno
hiredate > '01-JAN-88'
job IN ('PRESIDENT', 'CLERK', 'ANALYST')
sal BETWEEN 500 AND 1000
comm IS NULL AND sal = 2000
```

Conditions can have several forms, as shown in the following syntax. The description of each statement in Chapter 7, "SQL Statements", documents the restrictions on the conditions in the statement. The sections that follow describe the various forms of conditions.

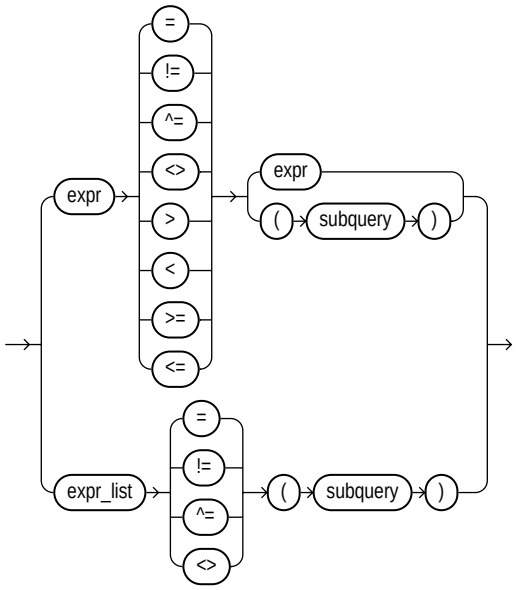
condition::=



Simple Comparison Conditions

A simple comparison condition specifies a comparison with expressions or subquery results.

simple_comparison_condition::=

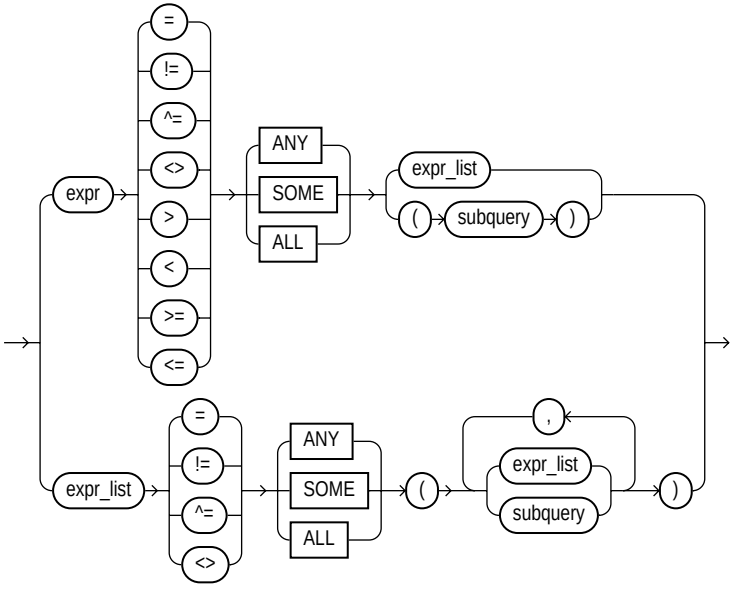


For information on comparison operators, see "Comparison Operators" on page 3-5.

Group Comparison Conditions

A group comparison condition specifies a comparison with any or all members in a list or subquery.

group_comparison_condition::=

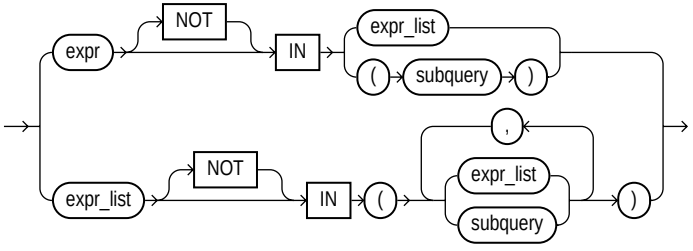


See "SELECT and Subqueries" on page 7-541.

Membership Conditions

A membership condition tests for membership in a list or subquery.

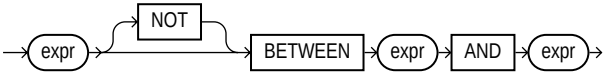
membership_condition::=



Range Conditions

A range condition tests for inclusion in a range.

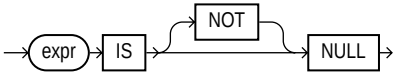
range_condition::=



NULL Conditions

A NULL condition tests for nulls.

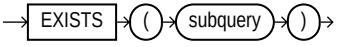
NULL_condition::=



EXISTS Conditions

An EXISTS condition tests for existence of rows in a subquery.

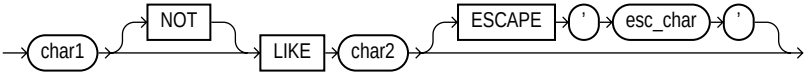
EXISTS_condition::=



LIKE Conditions

A LIKE condition specifies a test involving pattern matching.

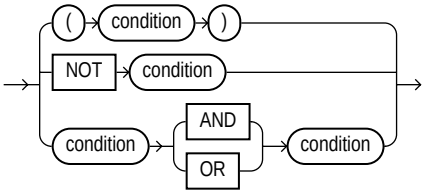
LIKE_condition::=



Compound Conditions

A compound condition specifies a combination of other conditions.

compound_condition::=



Queries and Subqueries

A **query** is an operation that retrieves data from one or more tables or views. In this reference, a top-level query is called a SELECT statement, and a query nested within a SELECT statement is called a **subquery**.

This section describes some types of queries and how to use them. The full syntax of all the clauses, and the semantics of the keywords and parameters, appear in "SELECT and Subqueries" on page 7-541.

Creating Simple Queries

The list of expressions that appears after the SELECT keyword and before the FROM clause is called the *select list*. Each expression *expr* becomes the name of one column in the set of returned rows, and each *table*.* becomes a set of columns, one for each column in the table in the order they were defined when the table was created. The datatype and length of each expression is determined by the elements of the expression.

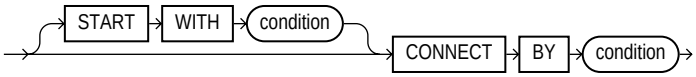
If two or more tables have some column names in common, you must qualify column names with names of tables. Otherwise, fully qualified column names are optional. However, it is always a good idea to qualify table and column references explicitly. Oracle often does less work with fully qualified table and column names.

You can use a column alias, *c_alias*, to label the preceding expression in the select list so that the column is displayed with a new heading. The alias effectively renames the select list item for the duration of the query. The alias can be used in the ORDER BY clause, but not other clauses in the query.

You can use comments in a SELECT statement to pass instructions, or *hints*, to the Oracle optimizer. The optimizer uses hints to choose an execution plan for the statement. For more information on hints, see "Hints" on page 2-58 and *Oracle8i Tuning*.

Hierarchical Queries

If a table contains hierarchical data, you can select rows in a hierarchical order using the hierarchical query clause:

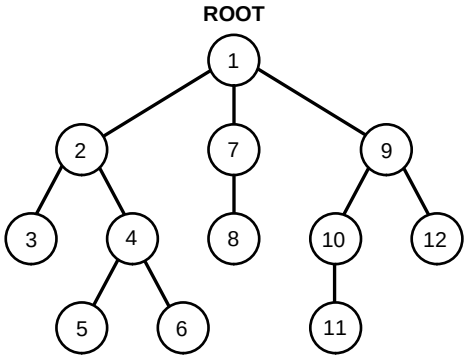


`START WITH`	specifies the root row(s) of the hierarchy.
`CONNECT BY`	specifies the relationship between parent rows and child rows of the hierarchy. Some part of *condition* must use the PRIOR operator to refer to the parent row. See the PRIOR operator on page 3-16.
`WHERE`	restricts the rows returned by the query without affecting other rows of the hierarchy.

Oracle uses the information from the hierarchical query clause clause to form the hierarchy using the following steps:

1. Oracle selects the root row(s) of the hierarchy—those rows that satisfy the START WITH condition.
2. Oracle selects the child rows of each root row. Each child row must satisfy the condition of the CONNECT BY condition with respect to one of the root rows.
3. Oracle selects successive generations of child rows. Oracle first selects the children of the rows returned in step 2, and then the children of those children, and so on. Oracle always selects children by evaluating the CONNECT BY condition with respect to a current parent row.
4. If the query contains a WHERE clause, Oracle eliminates all rows from the hierarchy that do not satisfy the condition of the WHERE clause. Oracle evaluates this condition for each row individually, rather than removing all the children of a row that does not satisfy the condition.
5. Oracle returns the rows in the order shown in Figure 5–1. In the diagram children appear below their parents.

Figure 5–1 Hierarchical Queries



To find the children of a parent row, Oracle evaluates the PRIOR expression of the CONNECT BY condition for the parent row and the other expression for each row in the table. Rows for which the condition is true are the children of the parent. The CONNECT BY condition can contain other conditions to further filter the rows selected by the query. The CONNECT BY condition cannot contain a subquery.

If the CONNECT BY condition results in a loop in the hierarchy, Oracle returns an error. A loop occurs if one row is both the parent (or grandparent or direct ancestor) and a child (or a grandchild or a direct descendent) of another row.

Sorting Query Results

You can use the ORDER BY clause to order the rows selected by a query. Sorting by position is useful in the following cases:

- To order by a lengthy select list expression, you can specify its position, rather than duplicate the entire expression, in the ORDER BY clause.
- For compound queries (containing set operators UNION, INTERSECT, MINUS, or UNION ALL), the ORDER BY clause must use positions, rather than explicit expressions. Also, the ORDER BY clause can appear only in the last component query. The ORDER BY clause orders all rows returned by the entire compound query.

The mechanism by which Oracle sorts values for the ORDER BY clause is specified either explicitly by the NLS_SORT initialization parameter or implicitly by the NLS_LANGUAGE initialization parameter. For information on these parameters, see *Oracle8i National Language Support Guide*. You can change the sort mechanism

dynamically from one linguistic sort sequence to another using the ALTER SESSION statement. You can also specify a specific sort sequence for a single query by using the NLSSORT function with the NLS_SORT parameter in the ORDER BY clause.

Joins

A **join** is a query that combines rows from two or more tables, views, or materialized views ("snapshots"). Oracle performs a join whenever multiple tables appear in the query's FROM clause. The query's select list can select any columns from any of these tables. If any two of these tables have a column name in common, you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

Join Conditions

Most join queries contain WHERE clause conditions that compare two columns, each from a different table. Such a condition is called a **join condition**. To execute a join, Oracle combines pairs of rows, each containing one row from each table, for which the join condition evaluates to TRUE. The columns in the join conditions need not also appear in the select list.

To execute a join of three or more tables, Oracle first joins two of the tables based on the join conditions comparing their columns and then joins the result to another table based on join conditions containing columns of the joined tables and the new table. Oracle continues this process until all tables are joined into the result. The optimizer determines the order in which Oracle joins tables based on the join conditions, indexes on the tables, and, in the case of the cost-based optimization approach, statistics for the tables.

In addition to join conditions, the WHERE clause of a join query can also contain other conditions that refer to columns of only one table. These conditions can further restrict the rows returned by the join query.

Equijoins

An **equijoin** is a join with a join condition containing an equality operator. An equijoin combines rows that have equivalent values for the specified columns. Depending on the internal algorithm the optimizer chooses to execute the join, the total size of the columns in the equijoin condition in a single table may be limited to the size of a data block minus some overhead. The size of a data block is specified by the initialization parameter DB_BLOCK_SIZE. See the "Equijoin Examples" on page 7-558.

Self Joins

A **self join** is a join of a table to itself. This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition. To perform a self join, Oracle combines and returns rows of the table that satisfy the join condition. See the "Self Join Example" on page 7-559.

Cartesian Products

If two tables in a join query have no join condition, Oracle returns their **Cartesian product**. Oracle combines each row of one table with each row of the other. A Cartesian product always generates many rows and is rarely useful. For example, the Cartesian product of two tables, each with 100 rows, has 10,000 rows. Always include a join condition unless you specifically need a Cartesian product. If a query joins three or more tables and you do not specify a join condition for a specific pair, the optimizer may choose a join order that avoids producing an intermediate Cartesian product.

Outer Joins

An outer join extends the result of a simple join. An **outer join** returns all rows that satisfy the join condition and those rows from one table for which no rows from the other satisfy the join condition. Such rows are not returned by a simple join. To write a query that performs an outer join of tables A and B and returns all rows from A, apply the outer join operator (+) to all columns of B in the join condition. For all rows in A that have no matching rows in B, Oracle returns NULL for any select list expressions containing columns of B. See the syntax for an outer join in "SELECT and Subqueries" on page 7-541.

Outer join queries are subject to the following rules and restrictions:

- The (+) operator can appear only in the WHERE clause or, in the context of left-correlation (that is, when specifying the TABLE clause) in the FROM clause, and can be applied only to a column of a table or view.
- If A and B are joined by multiple join conditions, you must use the (+) operator in all of these conditions. If you do not, Oracle will return only the rows resulting from a simple join, but without a warning or error to advise you that you do not have the results of an outer join.
- The (+) operator can be applied only to a column, not to an arbitrary expression. However, an arbitrary expression can contain a column marked with the (+) operator.

- A condition containing the (+) operator cannot be combined with another condition using the OR logical operator.
- A condition cannot use the IN comparison operator to compare a column marked with the (+) operator with an expression.
- A condition cannot compare any column marked with the (+) operator with a subquery.

If the WHERE clause contains a condition that compares a column from table B with a constant, the (+) operator must be applied to the column so that Oracle returns the rows from table A for which it has generated NULLs for this column. Otherwise Oracle will return only the results of a simple join.

In a query that performs outer joins of more than two pairs of tables, a single table can be the NULL-generated table for only one other table. For this reason, you cannot apply the (+) operator to columns of B in the join condition for A and B and the join condition for B and C.

Using Subqueries

Use subqueries for the following purposes:

- To define the set of rows to be inserted into the target table of an INSERT or CREATE TABLE statement
- To define the set of rows to be included in a view or materialized view ("snapshot) in a CREATE VIEW or CREATE MATERIALIZED VIEW statement
- To define one or more values to be assigned to existing rows in an UPDATE statement
- To provide values for conditions in a WHERE clause, HAVING clause, or START WITH clause of SELECT, UPDATE, and DELETE statements
- To provide values for a specified column in an INSERT ... VALUES list
- To provide values for arguments of a type constructor or a user-defined function
- To define a table to be operated on by a containing query.

  You do this by placing the subquery in the FROM clause of the containing query as you would a table name. You may use subqueries in place of tables in this way as well in INSERT, UDPATE, and DELETE statements.

  Subqueries so used can employ correlation variables, but only those defined within the subquery itself, not outer references. Outer references

("left-correlated subqueries") are allowed only in the FROM clause of a SELECT statement. See table_collection_expression on page 7-547.

A **subquery** answers multiple-part questions. For example, to determine who works in Taylor's department, you can first use a subquery to determine the department in which Taylor works. You can then answer the original question with the parent SELECT statement.

A subquery can contain another subquery. Oracle places no limit on the level of query nesting.

If tables in a subquery have the same name as tables in the containing statement, you must prefix any reference to the column of the table from the containing statement with the table name or alias. To make your statements easier for you to read, always qualify the columns in a subquery with the name or alias of the table, view, or materialized view.

Oracle performs a **correlated subquery** when the subquery references a column from a table referred to in the parent statement. A correlated subquery is evaluated once for each row processed by the parent statement. The parent statement can be a SELECT, UPDATE, or DELETE statement. See the "Correlated Subquery Examples" on page 7-566.

A correlated subquery answers a multiple-part question whose answer depends on the value in each row processed by the parent statement. For example, you can use a correlated subquery to determine which employees earn more than the average salaries for their departments. In this case, the correlated subquery specifically computes the average salary for each department.

Selecting from the DUAL Table

DUAL is a table automatically created by Oracle along with the data dictionary. DUAL is in the schema of the user SYS, but is accessible by the name DUAL to all users. It has one column, DUMMY, defined to be VARCHAR2(1), and contains one row with a value 'X'. Selecting from the DUAL table is useful for computing a constant expression with the SELECT statement. Because DUAL has only one row, the constant is returned only once. Alternatively, you can select a constant, pseudocolumn, or expression from any table, but the value will be returned as many times as there are rows in the table. See "SQL Functions" on page 4-1 for many examples of selecting a constant value from DUAL.

Distributed Queries

Oracle's distributed database management system architecture allows you to access data in remote databases using Net8 and an Oracle server. You can identify a remote table, view, or materialized view by appending *@dblink* to the end of its name. The *dblink* must be a complete or partial name for a database link to the database containing the remote table, view, or materialized view. For more information on referring to database links, see "Referring to Objects in Remote Databases" on page 2-74.

Distributed queries are currently subject to the restriction that all tables locked by a FOR UPDATE clause and all tables with LONG columns selected by the query must be located on the same database. For example, the following statement will raise an error:

```
SELECT emp_ny.*
    FROM emp_ny@ny, dept
    WHERE emp_ny.deptno = dept.deptno
    AND dept.dname = 'ACCOUNTING'
    FOR UPDATE OF emp_ny.sal;
```

The following statement fails because it selects LONG_COLUMN, a LONG value, from the EMP_REVIEW table on the NY database and locks the EMP table on the local database:

```
SELECT emp.empno, review.long_column, emp.sal
    FROM emp, emp_review@ny review
    WHERE emp.empno = emp_review.empno
    FOR UPDATE OF emp.sal;
```

6

About SQL Statements

. . . he is eminently plain and direct . . . both in his syntax and in his words . . . .

Matthew Arnold, *On Translating Homer*

This chapter describes the various types of Oracle SQL statements, and provides guidelines for finding the right SQL statement for your task. Topics include:

- Summary of SQL Statements
- Finding the Right SQL Statement

Summary of SQL Statements

The tables in the following sections provide a functional summary of SQL statements and are divided into these categories:

- Data Definition Language (DDL) Statements
- Data Manipulation Language (DML) Statements
- Transaction Control Statements
- Session Control Statements
- System Control Statements

Data Definition Language (DDL) Statements

Data definition language (DDL) statements enable you to perform these tasks:

- Create, alter, and drop schema objects
- Grant and revoke privileges and roles
- Analyze information on a table, index, or cluster

- Establish auditing options
- Add comments to the data dictionary

The CREATE, ALTER, and DROP commands require exclusive access to the specified object. For example, an ALTER TABLE statement fails if another user has an open transaction on the specified table.

The GRANT, REVOKE, ANALYZE, AUDIT, and COMMENT commands do not require exclusive access to the specified object. For example, you can analyze a table while other users are updating the table.

Oracle implicitly commits the current transaction before and after every DDL statement.

Many DDL statements may cause Oracle to recompile or reauthorize schema objects. For information on how Oracle recompiles and reauthorizes schema objects and the circumstances under which a DDL statement would cause this, see *Oracle8i Concepts*.

DDL statements are supported by PL/SQL with the use of the DBMS_SQL package. For more information, see *Oracle8i Supplied Packages Reference*.

Table 6–1 lists the DDL statements.

Table 6–1 Data Definition Language Statements

ALTER CLUSTER	CREATE DIMENSION	DROP DATABASE LINK
ALTER DATABASE	CREATE DIRECTORY	DROP DIMENSION
ALTER DIMENSION	CREATE FUNCTION	DROP DIRECTORY
ALTER FUNCTION	CREATE INDEX	DROP FUNCTION
ALTER INDEX	CREATE INDEXTYPE	DROP INDEX
ALTER MATERIALIZED VIEW / SNAPSHOT	CREATE LIBRARY	DROP INDEXTYPE
ALTER MATERIALIZED VIEW / SHAPSHOT LOG	CREATE MATERIALIZED VIEW / SHAPSHOT	DROP LIBRARY
ALTER PACKAGE	CREATE MATERIALIZED VIEW / SNAPSHOT LOG	DROP MATERIALIZED VIEW / SNAPSHOT
ALTER PROCEDURE	CREATE OPERATOR	DROP MATERIALIZED VIEW / SNAPSHOT LOG
ALTER PROFILE	CREATE PACKAGE	DROP OPERATOR
ALTER RESOURCE COST	CREATE PACKAGE BODY	DROP PACKAGE
ALTER ROLE	CREATE PROCEDURE	DROP PROCEDURE
ALTER ROLLBACK SEGMENT	CREATE PROFILE	DROP PROFILE
ALTER SEQUENCE	CREATE ROLE	DROP ROLE
ALTER SNAPSHOT	CREATE ROLLBACK SEGMENT	DROP ROLLBACK SEGMENT
ALTER SHAPSHOT LOG	CREATE SCHEMA	DROP SEQUENCE
ALTER TABLE	CREATE SEQUENCE	DROP SNAPSHOT
ALTER TABLESPACE	CREATE SHAPSHOT	DROP SNAPSHOT LOG
ALTER TRIGGER	CREATE SNAPSHOT LOG	DROP SYNONYM
ALTER TYPE	CREATE SYNONYM	DROP TABLE
ALTER USER	CREATE TABLE	DROP TABLESPACE
ALTER VIEW	CREATE TABLESPACE	DROP TRIGGER
ANALYZE	CREATE TEMPORARY TABLESPACE	DROP TYPE
ASSOCIATE STATISTICS	CREATE TRIGGER	DROP USER
AUDIT	CREATE TYPE	DROP VIEW
COMMENT	CREATE USER	GRANT
CREATE CLUSTER	CREATE VIEW	NOAUDIT
CREATE CONTEXT	DISASSOCIATE STATISTICS	RENAME
CREATE CONTROLFILE	DROP CLUSTER	REVOKE
CREATE DATABASE	DROP CONTEXT	TRUNCATE
CREATE DATABASE LINK		

Data Manipulation Language (DML) Statements

Data manipulation language (DML) statements query and manipulate data in existing schema objects. These statements do not implicitly commit the current transaction.

Table 6–2 Data Manipulation Language Statements

Statement
CALL
DELETE
EXPLAIN PLAN
INSERT
LOCK TABLE
SELECT
UPDATE

The CALL and EXPLAIN PLAN statements are supported in PL/SQL only when executed dynamically. All other DML statements are fully supported in PL/SQL.

Transaction Control Statements

Transaction control statements manage changes made by DML statements.

Table 6–3 Transaction Control Statements

Statement
COMMIT
ROLLBACK
SAVEPOINT
SET TRANSACTION

All transaction control statements except certain forms of the COMMIT and ROLLBACK commands are supported in PL/SQL. For information on the restrictions, see "COMMIT" on page 7-214 and "ROLLBACK" on page 7-537.

Session Control Statements

Session control statements dynamically manage the properties of a user session. These statements do not implicitly commit the current transaction.

PL/SQL does not support session control statements.

Table 6–4 Session Control Statements

Statement
`ALTER SESSION`
`SET ROLE`

System Control Statement

The single system control statement dynamically manages the properties of an Oracle instance. This statement does not implicitly commit the current transaction.

ALTER SYSTEM is not supported in PL/SQL.

Table 6–5 System Control Statement

Statement
`ALTER SYSTEM`

Embedded SQL Statements

Embedded SQL statements place DDL, DML, and transaction control statements within a procedural language program. Embedded SQL is supported by the Oracle precompilers and is documented in the following books:

- *Pro*COBOL Precompiler Programmer's Guide*
- *Pro*C/C++ Precompiler Programmer's Guide*
- *SQL*Module for Ada Programmer's Guide*

Finding the Right SQL Statement

The particular SQL statement you use to accomplish a given database task is sometimes obvious and sometimes difficult to predict. For example, you create a table with the CREATE TABLE statement. However, you don't enable a constraint with the ENABLE CONSTRAINT statement, because such a statement doesn't exist. Rather, you modify the column options using the ALTER TABLE statement.

This section lists, by database object and task, the appropriate SQL statement to use to accomplish various database tasks. You can then refer to Chapter 7, "SQL Statements", for the syntax and semantics of each SQL statement.

> **Note:** Your ability to use the SQL statements listed in this section depends on the version and edition of Oracle you are using, as well as the options you have installed. Be sure to read the detailed descriptions in Chapter 7, "SQL Statements", before using these statements.

Database Object / Task	Operation	SQL Statement
application	allowing to connect as a user	ALTER USER *proxy_clause*
application server	allowing to connect as a user	ALTER USER *proxy_clause*
auditing	of database events	CREATE TRIGGER
call	limit CPU time for	CPU_PER_CALL parameter
	limit data blocks read	LOGICAL_READS_PER_CALL parameter
checkpoint	perform explicitly	ALTER SYSTEM CHECKPOINT
clone database	mount	ALTER DATABASE MOUNT
cluster	cluster key, change columns of	prohibited
	extent, allocate for	ALTER CLUSTER *allocate_extent_clause*
	migrated or chained rows, identify	ANALYZE
	parallelism of, change	ALTER CLUSTER *parallel_clause*
	rename	prohibited
	storage characteristics of, change	ALTER CLUSTER *physical_attributes_clause*
	tablespace of, change	prohibited
	unused space in, release	ALTER CLUSTER *deallocate_unused_clause*
column	add to a table or modify	ALTER TABLE *add_column_options*, *modify_column_options*
	define	CREATE TABLE
	drop from a table	ALTER TABLE *drop_column_clause*

Database Object / Task	Operation	SQL Statement
	generate derived values automatically	CREATE TRIGGER
	organization of, define	CREATE TABLE
commit operation	prevent procedure or function from issuing	ALTER SESSION
compilation	avoid run-time of	ALTER FUNCTION ... COMPILE
constraint	add to a table or modify	ALTER TABLE *add_column_options*, *modify_column_options*
	business, enforce	CREATE TRIGGER
	enable, disable, or drop	ALTER TABLE *enable_disable_clause*, *drop_constraint_clause*
	specify	CREATE TABLE
control file	back up	ALTER DATABASE *controlfile_clauses*
	standby, create	ALTER DATABASE CREATE STANDBY CONTROLFILE
currency symbol	reset for session	ALTER SESSION SET NLS_CURRENCY
data	frequently used, caching	ALTER TABLE *cache_clause*
	specify as temporary or permanent	CREATE TABLE
data dictionary	convert from Oracle7 to Oracle8i	ALTER DATABASE CONVERT
data independence	provide	CREATE SYNONYM
database	character set of, change	ALTER DATABASE CHARACTER SET
	create script for	ALTER DATABASE *controlfile_clauses*
	database character set for, specify	CREATE DATABASE
	datafiles for, specify	CREATE DATABASE
	datafiles of, modify	ALTER DATABASE
	datafiles, establish number of	CREATE DATABASE
	downgrade to an earlier release	ALTER DATABASE RESET COMPATIBILITY
	global name of, change	ALTER DATABASE RENAME GLOBAL_NAME

Database Object / Task	Operation	SQL Statement
	global name resolution, enable for the session	ALTER SESSION SET GLOBAL_NAMES
	instances, establish number of	CREATE DATABASE
	media recovery, design	ALTER DATABASE *general_recovery_clause*
	media recovery, perform ongoing	ALTER DATABASE *managed_recovery_clause*
	mount	ALTER DATABASE MOUNT
	move a subset to a different Oracle database	ALTER TABLE *exchange_partition_clause*
	national character set for, specify	CREATE DATABASE
	national character set of, change	ALTER DATABASE CHARACTER SET
	open	ALTER DATABASE OPEN
	parallelize recovery of	ALTER DATABASE *parallel_clause*
	place in read-only mode	ALTER DATABASE OPEN
	place in read-write mode	ALTER DATABASE OPEN
	place in sustained standby recovery mode	ALTER DATABASE *general_recovery_clause*
	prepare to re-create	ALTER DATABASE *controlfile_clauses*
	recover	ALTER DATABASE *recover_clauses*
	redo log file groups, establish number of	CREATE DATABASE
	redo log files for, specify	CREATE DATABASE
	redo log files of, create or modify	ALTER DATABASE
	redo log files, establish number of	CREATE DATABASE
	redo log, choose mode for	CREATE DATABASE
	upgrade to Oracle8i	ALTER DATABASE
database character set	specify for a database	CREATE DATABASE
database events	transparent logging of	CREATE TRIGGER
database link	close	ALTER SESSION
database security	enforce authorizations	CREATE TRIGGER

Database Object / Task	Operation	SQL Statement
datafile	automatic extension of, allow	ALTER DATABASE DATAFILE *autoextend_clause*
	create	ALTER DATABASE CREATE DATAFILE
	put online	ALTER DATABASE DATAFILE ONLINE
	reconstruct damaged	ALTER DATABASE *general_recovery_clause*
	reconstruct lost or damaged	ALTER DATABASE CREATE DATAFILE
	recover specified	ALTER DATABASE *general_recovery_clause*
	replace an old, for recovery	ALTER DATABASE CREATE DATAFILE
	resize	ALTER DATABASE DATAFILE RESIZE
	take offline	ALTER DATABASE DATAFILE ONLINE / OFFLINE
	begin or end backup of	ALTER TABLESPACE ... BACKUP
	number of, establish for a database	CREATE DATABASE
	online, update instance information on	ALTER SYSTEM *check_datafiles_clause*
	specify for a database	CREATE DATABASE
dates	format of	See Table 2–9, "Date Format Elements" on page 2-40.
decimal character	reset for session	ALTER SESSION SET NLS_NUMERIC_ CHARACTERS
dimension	add a level, hierarchy, or attribute to	ALTER DIMENSION ... ADD
	change the relationships of	ALTER DIMENSION
	drop a level, hierarchy, or attribute from	ALTER DIMENSION ... DROP
	explicitly compile	ALTER DIMENSION ... COMPILE
dispatcher processes	multi-threaded server, manage	MTS_ parameters of ALTER SYSTEM
domain index	alter	ALTER INDEX ... PARAMETERS
	rebuild	ALTER INDEX *rebuild_clause*
dump file	limit the size of	ALTER SESSION SET MAX_DUMP_FILE_SIZE

Database Object / Task	Operation	SQL Statement
error messages	language in which displayed, change	ALTER SESSION SET NLS_LANGUAGE
function	allow to or prevent from committing a transaction	ALTER SESSION
	declaration of, change	CREATE OR REPLACE FUNCTION
	definition of, change	CREATE OR REPLACE FUNCTION
	recompile explicitly	ALTER FUNCTION
function-based index	disable	ALTER INDEX ... [*rebuild_clause*] DISABLE
	disabled, re-enable	ALTER INDEX ... [*rebuild_clause*] ENABLE
global names	enforce resolution of	GLOBAL_NAMES parameter of ALTER SYSTEM
hash join operations	data blocks for, allocate	ALTER SESSION SET HASH_MULTIBLOCK_IO_ COUNT
	in queries, enable or disable	ALTER SESSION SET HASH_JOIN_ENABLED ...
	memory for, allocate	ALTER SESSION SET HASH_AREA_SIZE
index	allow DML operations during rebuilding of	ALTER INDEX *rebuild_clause*
	based on a function; see "function-based index"	CREATE INDEX ... *column_expression*
	based on an indextype; see "domain index"	CREATE INDEX *domain_index_clause*
	collect statistics during rebuilding of	ALTER INDEX *rebuild_clause*
	default attribute values of, change	ALTER INDEX *partitioning_clauses*
	degree of parallelism for, change	ALTER INDEX *parallel_clause*
	direct-load INSERT operations, write to a log	ALTER INDEX *physical_attributes_clause*
	extent for, allocate new	ALTER INDEX *allocate_extent_clause*
	key compression, enable	ALTER INDEX *rebuild_clause*
	key values, eliminate repetition of	ALTER INDEX *rebuild_clause*
	merge block contents of	ALTER INDEX *rebuild_clause*

Database Object / Task	Operation	SQL Statement
	physical attributes of a partition of, change	ALTER INDEX *physical_attributes_clause*
	physical attributes of a subpartition of, change the	ALTER INDEX *physical_attributes_clause*
	physical attributes of, change	ALTER INDEX *physical_attributes_clause*
	re-create	ALTER INDEX *rebuild_clause*
	rebuild operations, write to a log	ALTER INDEX *rebuild_clause*
	SQL*Loader operations against, write to a log	ALTER INDEX *physical_attributes_clause*
	store bytes in reverse order	ALTER INDEX *rebuild_clause*
	tablespace for, specify	ALTER INDEX *rebuild_clause*
	tell Oracle not to use	ALTER INDEX ... [*rebuild_clause*] UNUSABLE
	unused space, release	ALTER INDEX *deallocate_unused_clause*
	rename	ALTER INDEX *rebuild_clause*
index partition	create-time attributes, change	ALTER INDEX *rebuild_clause*
	log direct-load INSERT operations	ALTER INDEX *physical_attributes_clause*
	log SQL*Loader operations against	ALTER INDEX *physical_attributes_clause*
	move to a different tablespace	ALTER INDEX *rebuild_clause*
	physical attributes of, change	ALTER INDEX *physical_attributes_clause*
	physical, logging, or storage characteristics of, change	ALTER INDEX *partitioning_clauses*
	re-create	ALTER INDEX *rebuild_clause*
	remove from the database	ALTER INDEX *partitioning_clauses*
	specify a tablespace for	ALTER INDEX *rebuild_clause*
	split into two partitions	ALTER INDEX *partitioning_clauses*
	tell Oracle not to use	ALTER INDEX ... UNUSABLE
index subpartition	change a create-time attributes, change	ALTER INDEX *rebuild_clause*
	log direct-load INSERT operations	ALTER INDEX *physical_attributes_clause*

Database Object / Task	Operation	SQL Statement
	log SQL*Loader operations against	ALTER INDEX *physical_attributes_clause*
	move to a different tablespace	ALTER INDEX *rebuild_clause*
	physical attributes, change	ALTER INDEX *physical_attributes_clause*
	physical, logging, or storage characteristics, change	ALTER INDEX *partitioning_clauses*
	re-create	ALTER INDEX *rebuild_clause*
	tablespace for, specify	ALTER INDEX *rebuild_clause*
	tell Oracle not to use	ALTER INDEX ... UNUSABLE
index-organized table	characteristics, change	ALTER TABLE
indexes	on a cluster	CREATE INDEX
	on a nested table storage table	CREATE INDEX
	on a partitioned table	CREATE INDEX
	on an index-organized table	CREATE INDEX
	on columns of a table	CREATE INDEX
	on scalar typed object attributes	CREATE INDEX
instance	dynamically modify	ALTER SYSTEM
	make an index extent available to	ALTER INDEX *allocate_extent_clause*
	switch to a different	ALTER SESSION SET INSTANCE
instance recovery	continue after interruption	ALTER DATABASE *general_recovery_clause*
instances	number of, establish for a database	CREATE DATABASE
Java class	force resolution of	ALTER JAVA
Java resource	force compilation of	ALTER JAVA
Java source	force compilation of	ALTER JAVA
licensing	changing limits or thresholds	LICENSE_ parameters of ALTER SYSTEM
LOB columns	add to a table or modify	ALTER TABLE *add_column_options*, *modify_column_options*, *LOB_storage_clause*
location transparency	provide	CREATE SYNONYM

Database Object / Task	Operation	SQL Statement
materialized view	automatic refresh, change the mode or timing of	ALTER MATERIALIZED VIEW *refresh_clause*
	change from rowid-based to primary-key-based	ALTER MATERIALIZED VIEW ALTER MATERIALIZED VIEW LOG
	degree of parallelism, specify or change	ALTER MATERIALIZED VIEW *parallel_clause*
	divide into partitions	ALTER MATERIALIZED VIEW *partitioning_clauses*
	LOB storage characteristics, change	ALTER MATERIALIZED VIEW *modify_LOB_storage_clause*
	LOB storage characteristics, specify	ALTER MATERIALIZED VIEW *LOB_storage_clause*
	log changes to	ALTER MATERIALIZED VIEW ... LOGGING
	make eligible for query rewrite	ALTER MATERIALIZED VIEW ... QUERY REWRITE ALTER SESSION SET QUERY_REWRITE_ENABLED
	make frequently accessed data accessible	ALTER MATERIALIZED VIEW ... CACHE
	revalidate	ALTER MATERIALIZED VIEW ... COMPILE
	storage characteristics, change	ALTER MATERIALIZED VIEW *physical_attributes_clause*
materialized view log	automatic refresh, change the mode and timing of	ALTER MATERIALIZED VIEW LOG
	change from rowid-based to primary-key-based	ALTER MATERIALIZED VIEW LOG
	divide into partitions	ALTER MATERIALIZED VIEW LOG *partitioning_clauses*
	physical and storage characteristics, change	ALTER MATERIALIZED VIEW LOG ... *physical_attributes_clause*
	save both old and new values	ALTER MATERIALIZED VIEW LOG ...NEW VALUES
	store primary key of changed rows	ALTER MATERIALIZED VIEW LOG ... ADD

Database Object / Task	Operation	SQL Statement
	store rowid of changed rows	ALTER MATERIALIZED VIEW LOG ... ADD
media recovery	avoid on startup	ALTER DATABASE DATAFILE END BACKUP
	from specified redo log file	ALTER DATABASE *general_recovery_clause*
	prepare for	ALTER DATABASE ARCHIVELOG
national character set	specify for a database	CREATE DATABASE
national language support	change settings for the session	ALTER SESSION SET NLS_ parameters
nested table	update in a view	create an `INSTEAD OF` trigger
nested table columns	indexing	CREATE INDEX
numbers	format	See Table 2–7, "Number Format Elements" on page 2-36.
object references. See REFs		
online redo log	reinitialize	ALTER DATABASE CLEAR LOGFILE
outline	assign to a different category	ALTER OUTLINE ... CHANGE CATEGORY TO
	recompile	ALTER OUTLINE ... REBUILD
	rename	ALTER OUTLINE ... RENAME
	automatically create and store	ALTER SESSION SET CREATE_STORED_ OUTLINES
	use to generate execution plans	ALTER SESSION SET USE_STORED_OUTLINES
package	avoid run-time compilation	ALTER PACKAGE
	compile explicitly	ALTER PACKAGE
package body	avoid run-time compilation	ALTER PACKAGE
	recompile explicitly	ALTER PACKAGE
parallelism	specify for a table	CREATE TABLE
	specify for DML on a table	CREATE TABLE
parameter, initialization	change the setting for the current session	ALTER SESSION *set_clause*
parameter, session	set or change the setting of	ALTER SESSION *set_clause*
partition	add to a table or modify	ALTER TABLE

Database Object / Task	Operation	SQL Statement
	default attributes, change	ALTER TABLE *modify_default_attributes_clause*
	logging characteristics, change	ALTER TABLE *logging_clause*
	merge with another partition	ALTER TABLE *merge_partitions_clause*
	point to data in a nonpartitioned table	ALTER TABLE *exchange_partition_clause*
	real attributes, change	ALTER TABLE *modify_partition_clause*
password	complexity of, guarantee	PASSWORD_VERIFY_FUNCTION parameter
	make unavailable	PASSWORD_REUSE_TIME parameter
	number of days account will be locked after failed login attempts, specify	PASSWORD_LOCK_TIME parameter
	number of days before reuse, limit	PASSWORD_REUSE_TIME parameter
	number of days in grace period, specify	PASSWORD_GRACE_TIME parameter
	number of days usable, limit	PASSWORD_LIFE_TIME parameter
	number of times reused, limit	PASSWORD_REUSE_MAX parameter
	special characters in, allow	PASSWORD_VERIFY_FUNCTION parameter
performance	optimize for index access path	ALTER SESSION SET OPTIMIZER_INDEX_ COST_ADJ
	optimize for nested loop joins	ALTER SESSION SET OPTIMIZER_INDEX_ CACHING
	specify an optimizer search limit	ALTER SESSION SET OPTIMIZER_SEARCH_ LIMIT
	specify the optimizer approach for the session	ALTER SESSION SET OPTIMIZER_MODE
procedure	allow to or prevent from committing a transaction	ALTER SESSION
	avoid run-time compilation	ALTER PROCEDURE
	recompile explicitly	ALTER PROCEDURE
profile	resource limit, add to	ALTER PROFILE
	resource limit, change	ALTER PROFILE

Database Object / Task	Operation	SQL Statement
	resource limit, drop from	ALTER PROFILE
recovery	distributed, enable or disable	ALTER SYSTEM *distributed_recovery_clause*
recovery data	discard	ALTER DATABASE RESETLOGS
redo log	remove changes from	ALTER DATABASE OPEN RESETLOGS
	reset sequence of	ALTER DATABASE OPEN RESETLOGS
	specify mode of	CREATE DATABASE
redo log file	add	ALTER DATABASE ADD LOGFILE MEMBER
	automatically generates names for	ALTER DATABASE *general_recovery_clause*
	clear	ALTER DATABASE CLEAR LOGFILE
	drop	ALTER DATABASE DROP LOGFILE
	enable or disable thread	ALTER DATABASE ENABLE THREAD
	rename	ALTER DATABASE RENAME FILE
	number of, establish for a database	CREATE DATABASE
	archive manually or automatically	ALTER SYSETM *archive_log_clause*
	number of, establish for a database	CREATE DATABASE
	specify a path for	ALTER SESSION SET LOG_ARCHIVE_DEST_*n*
	switch manually	ALTER SYSTEM *switch_logfile_clause*
REFS	validate and update	ANALYZE
role	change authorization required	ALTER ROLE
rollback segment	bring online	ALTER ROLLBACK SEGMENT
	reduce in size	ALTER ROLLBACK SEGMENT
	storage characteristics, change	ALTER ROLLBACK SEGMENT
	take offline	ALTER ROLLBACK SEGMENT
rowid	examine	query the ROWID pseudocolumn
	extended, interpreting contents	DBMS_ROWID package; see *Oracle8i Supplied Packages Reference*

Database Object / Task	Operation	SQL Statement
schema	change during the session	ALTER SESSION SET CURRENT_SCHEMA
schema object	reference without referencing its location	CREATE SYNONYM
	reference without referencing its owner	CREATE SYNONYM
	specify another name for	CREATE SYNONYM
	validate structure of	ANALYZE
sequence	cached sequence values, change number of	ALTER SEQUENCE *cache_clause*
	consecutive order of values, guarantee	CREATE SEQUENCE ... ORDER ALTER SEQUENCE ... ORDER
	create	CREATE SEQUENCE
	determine current value of	See "CURRVAL and NEXTVAL" on page 2-51.
	increment value, set	CREATE SEQUENCE ... INCREMENT BY ALTER SEQUENCE ... INCREMENT BY
	maximum or minimum value, eliminate	ALTER SEQUENCE
	minimum or maximum value, set	CREATE SEQUENCE ALTER SEQUENCE
	preallocate values for faster access	CREATE SEQUENCE ALTER SEQUENCE
	restart after a predefined limit	CREATE SEQUENCE ... CYCLE ALTER SEQUENCE ... CYCLE
	starting value, set	CREATE SEQUENCE
server processes	multi-threaded server, manage	MTS_ parameters of ALTER SYSTEM
session	CPU time for, limit	CPU_PER_SESSION parameter
	data blocks read, limit	LOGICAL_READS_PER_SESSION parameter
	enable or disable parallel transactions in	ALTER SESSION
	inactive period duration, limit	IDLE_TIME parameter

Database Object / Task	Operation	SQL Statement
	private SGA space for, limit	PRIVATE_SGA parameter
	resource costs allowed, change	ALTER RESOURCE COST
	restrict to privileged users	ALTER SYSTEM *restricted_session_clause*
	terminate	ALTER SYSTEM *kill_session_clause*
	total elapsed time, limit	CONNECT_TIME parameter
	total resources for, limit	COMPOSITE_LIMIT parameter
SGA	flush data from shared pool	ALTER SYSTEM *flush_shared_pool_clause*
shared pool	flush	ALTER SYSTEM *flush_shared_pool_clause*
snapshot. *See* "**materialized view**".		
sort operations	linguistic sequence, change	ALTER SESSION SET NLS_SORT
standby database	activate	ALTER DATABASE ACTIVATE STANDBY DATABASE
	recover	ALTER DATABASE *recover_clauses*
statistics	on a schema object, collect	ANALYZE
	on a schema object, delete	ANALYZE
	on scalar object attributes, collect	ANALYZE
subpartition	add to a table or modify	ALTER TABLE
	default attributes, change	ALTER TABLE *modify_default_attributes_clause, modify_partition_clause*
	logging characteristics, change	ALTER TABLE *logging_clause*
	real attributes, change	ALTER TABLE *modify_subpartition_clause*
system resources	enable or disable	RESOURCE_LIMITS parameter of ALTER SYSTEM
table	allocate space for	ALTER TABLE *allocate_extent_clause*
	characteristics, change	ALTER TABLE *physical_attributes_clause, storage_clauses*
	column, drop from table	ALTER TABLE *drop_column_clause*
	degree of parallelism, change	ALTER TABLE *parallel_clause*
	logging characteristics, change	ALTER TABLE *logging_clause*

Database Object / Task	Operation	SQL Statement
	make read-only, read-write	ALTER TABLE
	migrated or chained rows, identify	ANALYZE
	organization, define	CREATE TABLE
	partition, point to the contents of another table	ALTER TABLE *exchange_partition_clause*
	partitioning, specify	CREATE TABLE
	rename	ALTER TABLE
	unused space of, release	ALTER TABLE *deallocate_unused_clause*
	heap or index organized	CREATE TABLE
	include in a cluster	CREATE TABLE
	replicate asynchronous, maintain	CREATE TRIGGER
	storage characteristics of, set	CREATE TABLE
tablespace	allow or disallow writing to	ALTER TABLESPACE READ WRITE / ONLY
	datafiles, add or rename	ALTER TABLESPACE *datafile/tempfile_clauses*
	logging characteristics, change	ALTER TABLESPACE
	minimum extent length, change	ALTER TABLESPACE
	reconstruct damaged	ALTER DATABASE *general_recovery_clause*
	reconstruct lost or damaged	ALTER DATABASE CREATE DATAFILE
	recover specified	ALTER DATABASE *general_recovery_clause*
	specifying for a table	CREATE TABLE
	storage characteristics, change	ALTER TABLESPACE
	take online or offline	ALTER TABLESPACE
	user quota on, change	ALTER USER
	assign to a user	CREATE USER
	space quota for a user, allocate	CREATE USER
tempfile	allow for automatic extension of	ALTER DATABASE TEMPFILE
	resize	ALTER DATABASE TEMPFILE

Database Object / Task	Operation	SQL Statement
transaction	distributed, force commit of	ALTER SESSION
	distributed, force rollback of	ALTER SESSION
trigger	enable or disable	ALTER TABLE
user	authentication, change	ALTER USER
	database resources limits, change	ALTER USER *profile_clause*
	default roles, change	ALTER USER
	failed attempts to log in, limit	FAILED_LOGIN_ATTEMPTS parameter
	number of sessions, limit	SESSIONS_PER_USER parameter
	password, change	ALTER USER
	resource limits, set	CREATE USER
	restrict access to Oracle	ALTER SYSTEM *restricted_session_clause*
	tablespace quota, allocate	CREATE USER
	tablespaces, assign	CREATE USER

7

SQL Statements

A whole is that which has a beginning, a middle, and an end.

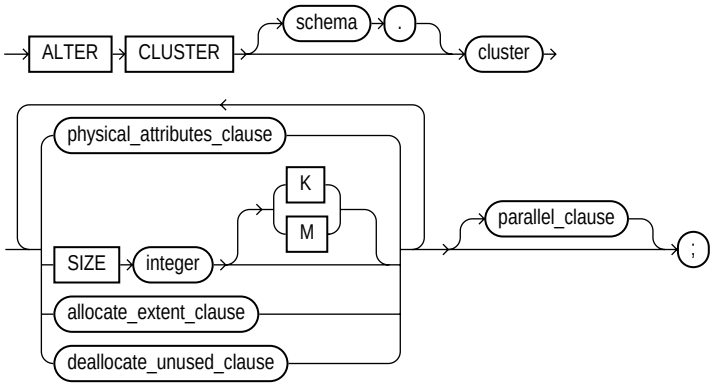
Aristotle, *Poetics*

This chapter describes, in alphabetical order, Oracle SQL statements and major clauses. The description of each statement or clause contains the following sections:

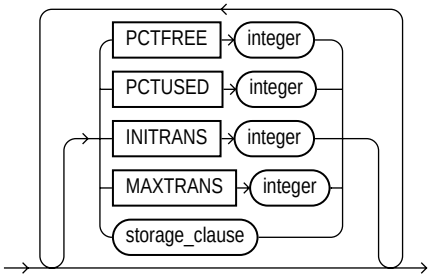
Syntax	shows the keywords and parameters that make up the statement. **Caution: Not all keywords and parameters are valid in all circumstances. Be sure to refer to the "Keywords and Parameters" section of each statement and clause to learn about any restrictions on the syntax.**
Purpose	describes the basic uses of the statement.
Prerequisites	lists privileges you must have and steps that you must take before using the statement. In addition to the prerequisites listed, most statements also require that the database be opened by your instance, unless otherwise noted.
Keywords and Parameters	describes the purpose of each keyword and parameter. (The conventions for keywords and parameters used in this chapter are explained in the Preface of this reference.) Restrictions and usage notes also appear in this section.
Examples	shows how to use various clauses and parameters of the statement.

ALTER CLUSTER

Syntax

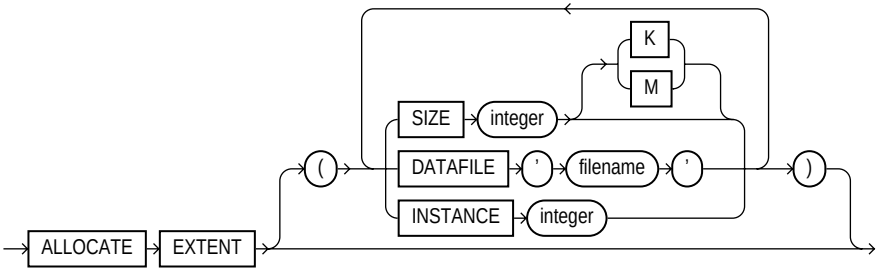


physical_attributes_clause::=

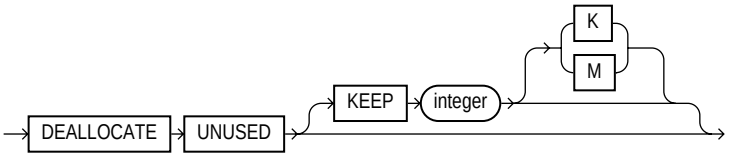


storage_clause: See "storage_clause" on page 7-575.

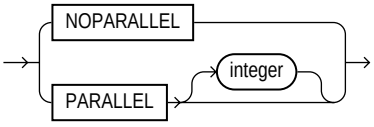
allocate_extent_clause::=



deallocate_unused_clause::=



parallel_clause::=



Purpose

To redefine storage and parallelism characteristics of a cluster.

For information on creating a cluster, see "CREATE CLUSTER" on page 7-236.

To remove tables from a cluster, see "DROP CLUSTER" on page 7-446 and "DROP TABLE" on page 7-475.

> **Note:** You cannot use this statement to change the number or the name of columns in the cluster key, and you cannot change the tablespace in which the cluster is stored.

Prerequisites

The cluster must be in your own schema or you must have ALTER ANY CLUSTER system privilege.

Keywords and Parameters

schema is the schema containing the cluster. If you omit *schema*, Oracle assumes the cluster is in your own schema.

cluster is the name of the cluster to be altered.

physical_attributes_clause changes the values of the PCTUSED, PCTFREE, INITRANS, and MAXTRANS parameters of the cluster. For a description of these parameters, see "CREATE CLUSTER" on page 7-236.

storage_clause changes the storage characteristics for the cluster. See the "storage_clause" on page 7-575.

Restriction: You cannot change the values of the storage parameters INITIAL and MINEXTENTS for a cluster.

SIZE *integer* determines how many cluster keys will be stored in data blocks allocated to the cluster. For a description of the SIZE parameter, see "CREATE CLUSTER" on page 7-236.

Restriction: You can change the SIZE parameter only for an indexed cluster, not for a hash cluster.

allocate_extent_clause explicitly allocates a new extent for the cluster.

Restriction: You can allocate a new extent only for an indexed cluster, not for a hash cluster.

SIZE specifies the size of the extent in bytes. Use K or M to specify the extent size in kilobytes or megabytes.

When you explicitly allocate an extent with this clause, Oracle does not evaluate the cluster's storage parameters and determine a new size for the next extent to be allocated (as it does when you create a table). Therefore, specify SIZE if you do not want Oracle to use a default value.

DATAFILE specifies one of the datafiles in the cluster's tablespace to contain the new extent. If you omit this parameter, Oracle chooses the datafile.

INSTANCE makes the new extent available to the specified instance. An instance is identified by the value of its initialization parameter INSTANCE_NUMBER. If you omit INSTANCE, the extent is available to all instances. **Use this parameter only if you are using Oracle with the Parallel Server option in parallel mode.**

deallocate_unused_clause explicitly deallocates unused space at the end of the cluster and makes the freed space available for other segments. Only unused space above the high water mark can be freed.

- KEEP — specifies the number of bytes above the high water mark that the cluster will have after deallocation. If the number of remaining extents is less than MINEXTENTS, then MINEXTENTS is set to the current number of extents. If the initial extent becomes smaller than INITIAL, then INITIAL is set to the value of the current initial extent. If you omit KEEP, all unused space is freed.

For a more complete description of this clause, see "ALTER TABLE" on page 7-113.

parallel_clause changes the default degree of parallelism for queries and DML on the cluster. For more detailed information, see the Notes to the *parallel_clause* of "CREATE TABLE" on page 7-359.

- NOPARALLEL — specifies serial execution. This is the default.
- PARALLEL — causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.
- PARALLEL *integer* — specifies the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution processes. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Restriction: If the tables in *cluster* contain any columns of LOB or user-defined object type, this statement as well as subsequent INSERT, UPDATE, or DELETE operations on *cluster* are executed serially without notification.

Examples

The following statement alters the CUSTOMER cluster in the schema SCOTT:

```
ALTER CLUSTER scott.customer
    SIZE 512
    STORAGE (MAXEXTENTS 25);
```

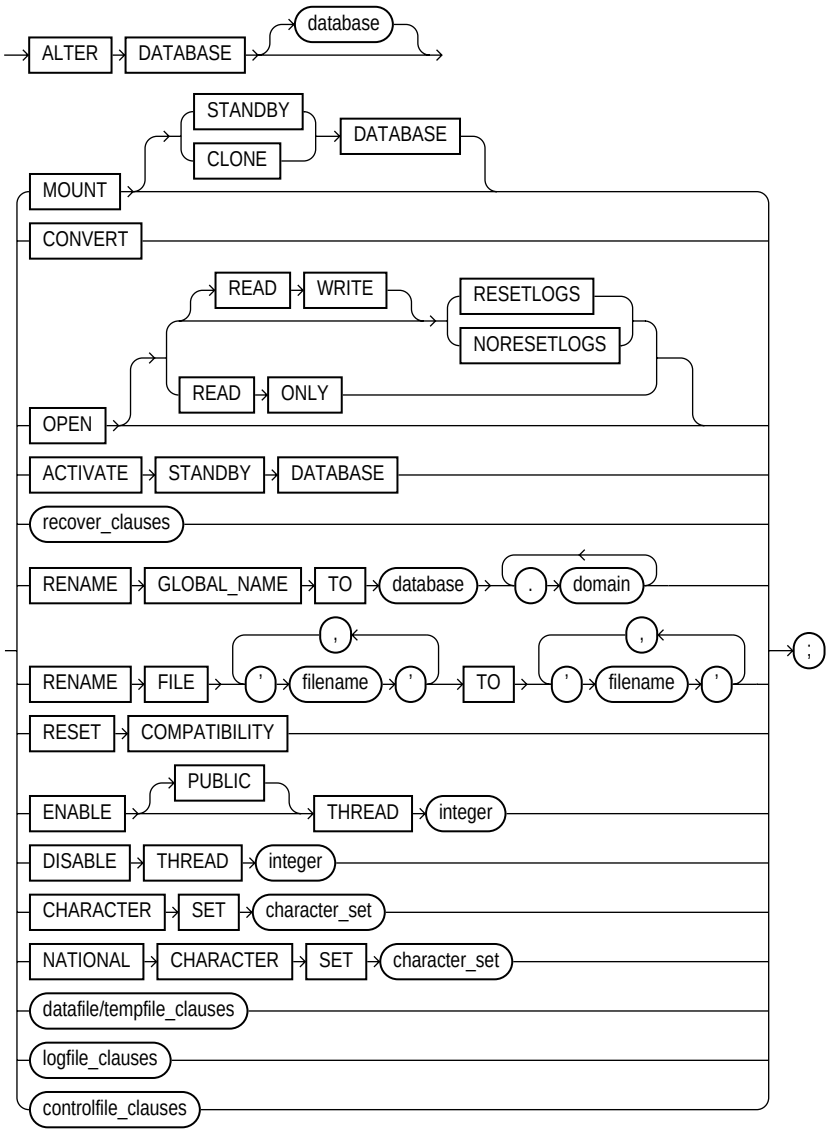
Oracle allocates 512 bytes for each cluster key value. Assuming a data block size of 2 kilobytes, future data blocks within this cluster contain 4 cluster keys per data block, or 2 kilobytes divided by 512 bytes. The cluster can have a maximum of 25 extents.

The following statement deallocates unused space from the CUSTOMER cluster, keeping 30 kilobytes of unused space for future use:

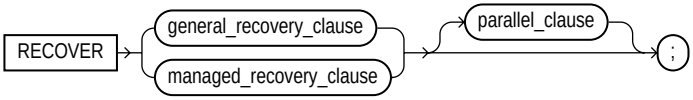
```
ALTER CLUSTER scott.customer
    DEALLOCATE UNUSED KEEP 30 K;
```

ALTER DATABASE

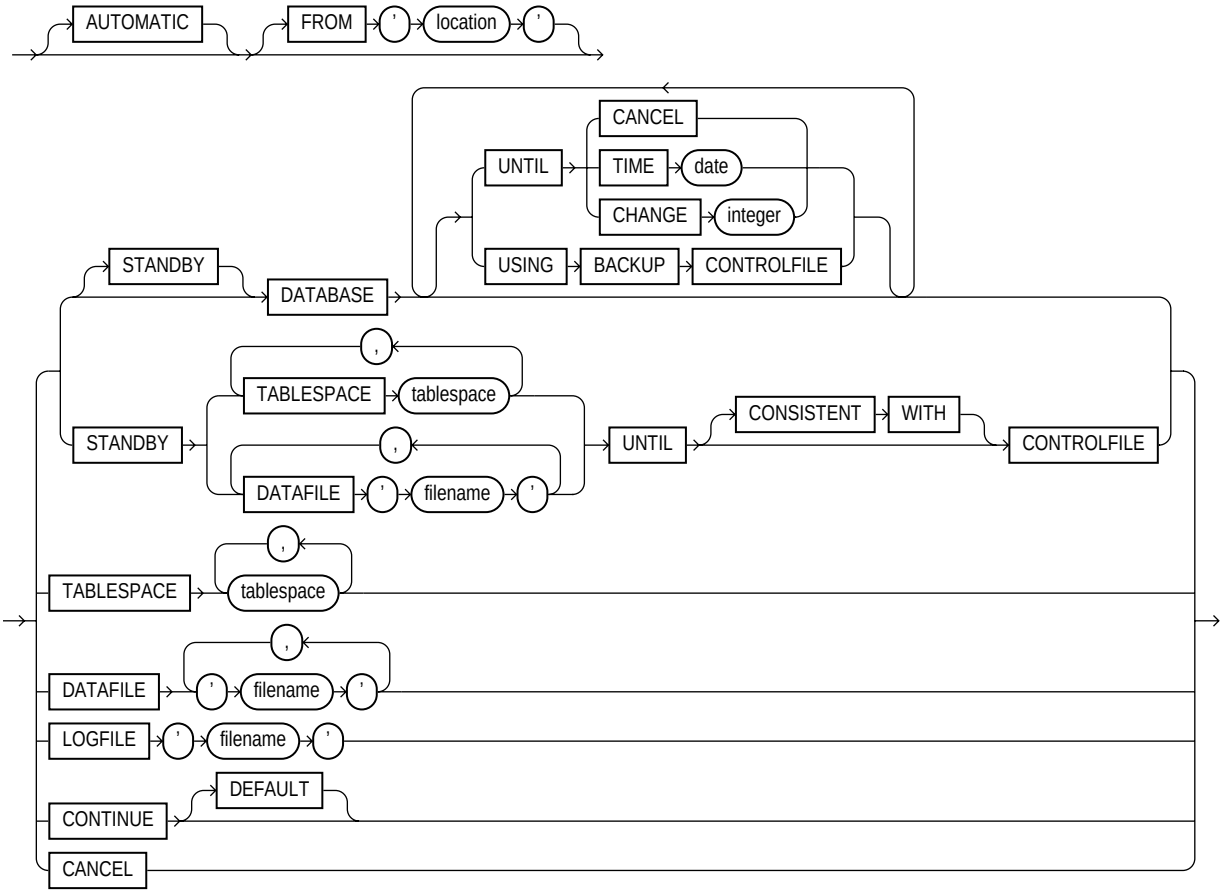
Syntax



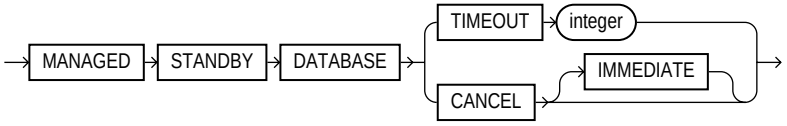
recover_clauses::=



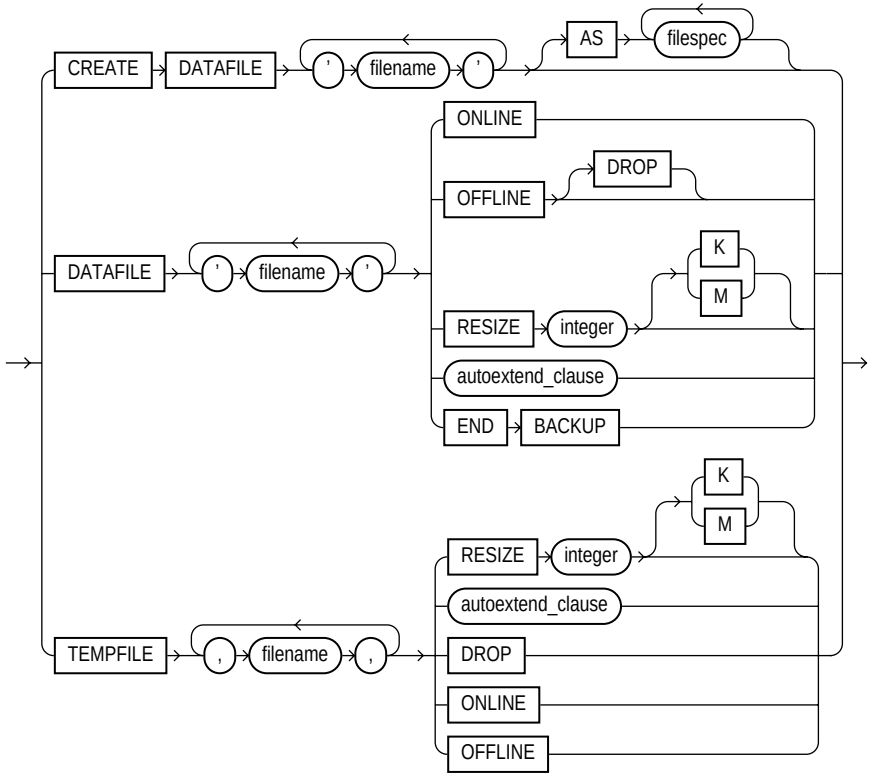
general_recovery_clause::=



managed_recovery_clause::=

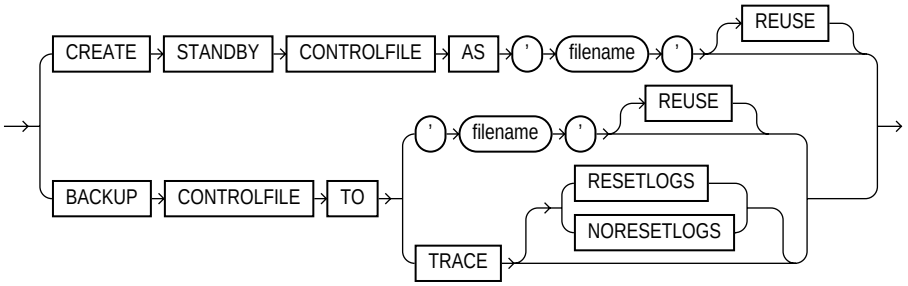


datafile/tempfile_clauses::=

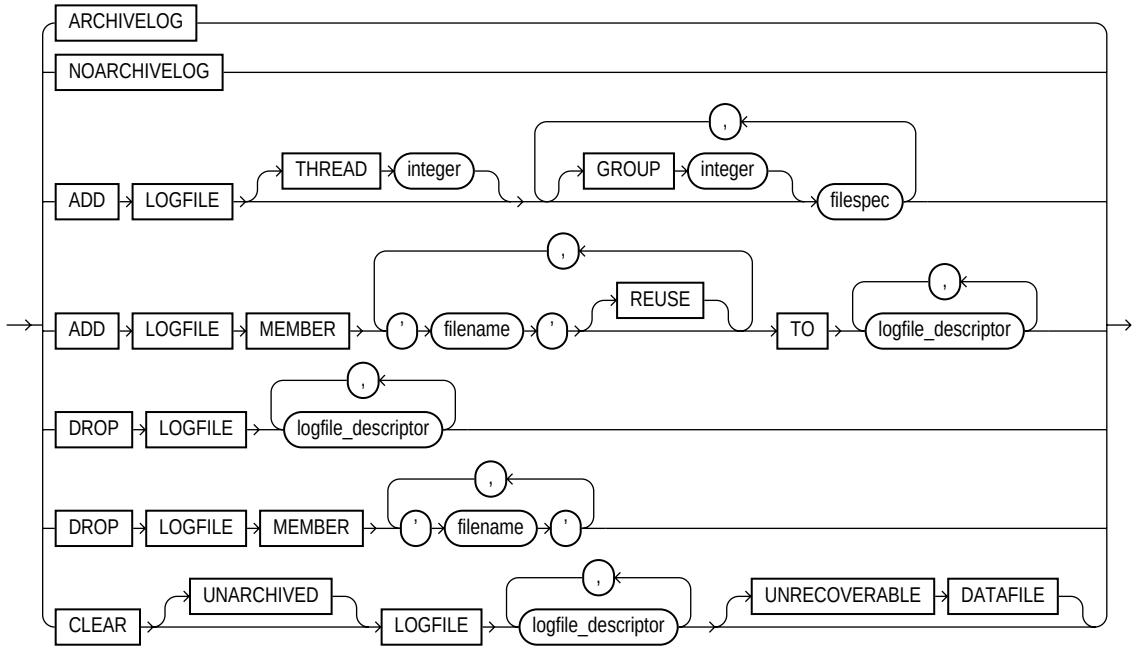


filespec: See "filespec" on page 7-490.

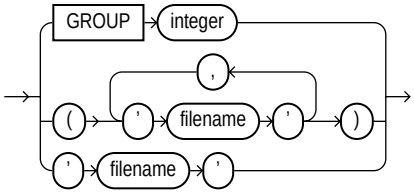
controlfile_clauses::=



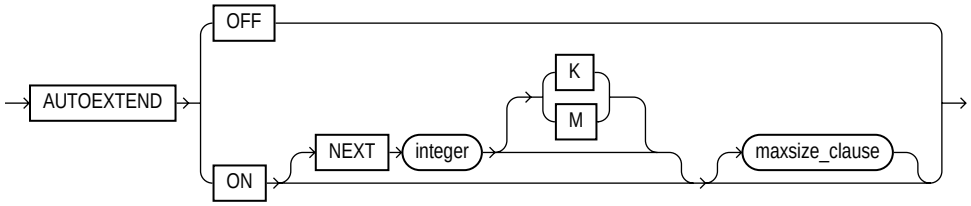
logfile_clauses::=



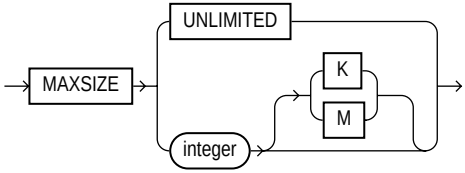
logfile_descriptor::=



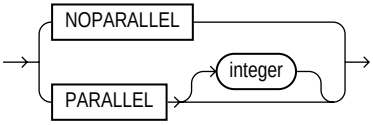
autoextend_clause::=



maxsize_clause::=



parallel_clause::=



Purpose

To modify, maintain, or recover an existing database.

For more information on using the ALTER DATABASE statement for database maintenance, see the *Oracle8i Administrator's Guide*.

For examples of performing media recovery, see *Oracle8i Administrator's Guide* and *Oracle8i Backup and Recovery Guide*.

For information on creating a database, see "CREATE DATABASE" on page 7-249.

Prerequisites

You must have ALTER DATABASE system privilege.

To specify the RECOVER clause, you must also have the OSDBA role enabled.

Keywords and Parameters

database identifies the database to be altered. The database name can contain only ASCII characters. If you omit database, Oracle alters the database identified by the value of the initialization parameter DB_NAME. You can alter only the database whose control files are specified by the initialization parameter CONTROL_FILES. The database identifier is not related to the Net8 database specification.

You can use the following clauses only when the database is not mounted by your instance:

MOUNT mounts the database.

- STANDBY DATABASE mounts the standby database. For more information, see the *Oracle8i Backup and Recovery Guide*.
- CLONE DATABASE mounts the clone database. For more information, see the *Oracle8i Backup and Recovery Guide*.

CONVERT completes the conversion of the Oracle7 data dictionary. After you use this clause, the Oracle7 data dictionary no longer exists in the Oracle database. **Use this clause only when you are migrating to Oracle8*i*.** For more information, see *Oracle8i Migration*.

ACTIVATE STANDBY DATABASE changes the state of a standby database to an active database. For more information, see *Oracle8i Backup and Recovery Guide*.

OPEN opens the database, making it available for normal use. You must mount the database before you can open it. You must activate a standby database before you can open it.

- READ ONLY restricts users to read-only transactions, preventing them from generating redo logs. You can use this clause to make a standby database available for queries even while archive logs are being copied from the primary database site.

Restrictions:

- You cannot open a database READ ONLY if it is currently opened READ WRITE by another instance.
- You cannot open a database READ ONLY if it requires recovery.
- You cannot take tablespaces offline while the database is open READ ONLY. However, you can take datafiles offline and online, and you can recover offline datafiles and tablespaces while the database is open READ ONLY.

READ WRITE — opens the database in read-write mode, allowing users to generate redo logs. This is the default.

RESETLOGS — resets the current log sequence number to 1 and discards any redo information that was not applied during recovery, ensuring that it will never be applied. This effectively discards all changes that are in the redo log, but not in the database. You must use this clause to open the database after performing media recovery with an incomplete recovery using the RECOVER clause or with a backup control file. After opening the database with this clause, you should perform a complete database backup.

NORESETLOGS — leaves the log sequence number and redo log files in their current state.

Restriction: You can specify RESETLOGS and NORESETLOGS only after performing incomplete media recovery or complete media recovery with a backup control file. In any other case, Oracle uses the NORESETLOGS automatically.

You can use any of the following clauses when your instance has the database mounted, open or closed, and the files involved are not in use:

general_recovery_clause — lets you design media recovery for the database or standby database, or for specified tablespaces or files. For more information on media recovery, see *Oracle8i Backup and Recovery Guide*.

Note: If you do not have special media requirements, Oracle Corporation recommends that you use the SQL*Plus RECOVER statement. For more information, see *SQL*Plus User's Guide and Reference*.

Restrictions:

- You can recover the entire database only when the database is closed.
- Your instance must have the database mounted in exclusive mode.
- You can recover tablespaces or datafiles when the database is open or closed, provided that the tablespaces or datafiles to be recovered are offline.
- You cannot perform media recovery if you are connected to Oracle through the multi-threaded server architecture.

AUTOMATIC — automatically generates the name of the next archived redo log file needed to continue the recovery operation. Oracle uses the LOG_ARCHIVE_DEST (or LOG_ARCHIVE_DEST_1) and LOG_ARCHIVE_FORMAT parameters (or their defaults) to generate the target redo log filename. If the file is found, the redo contained in that file is applied. If the file is not found, Oracle prompts you for a filename, displaying the generated filename as a suggestion.

If you specify neither AUTOMATIC nor LOGFILE, Oracle prompts you for a filename, displaying the generated filename as a suggestion. You can then accept the generated filename or replace it with a fully qualified filename. If you know the archived filename differs from what Oracle would generate, you can save time by using the LOGFILE clause.

FROM '*location*' — specifies the location from which the archived redo log file group is read. The value of *location* must be a fully specified file location following the conventions of your operating system. If you omit this parameter, Oracle assumes the archived redo log file group is in the location specified by the initialization parameter LOG_ARCHIVE_DEST or LOG_ARCHIVE_DEST_1.

STANDBY DATABASE — recovers the standby database using the control file and archived redo log files copied from the primary database. The standby database must be mounted but not open.

DATABASE — recovers the entire database. This is the default. You can use this clause only when the database is closed.

Note: This clause recovers only online datafiles.

UNTIL — specifies the duration of the recovery operation.

- CANCEL performs cancel-based recovery. This clause recovers the database until you issue the ALTER DATABASE RECOVER statement with the RECOVER CANCEL clause.
- TIME performs time-based recovery. This parameter recovers the database to the time specified by the date. The date must be a character literal in the format 'YYYY-MM-DD:HH24:MI:SS'.
- CHANGE performs change-based recovery. This parameter recovers the database to a transaction-consistent state immediately before the system change number (SCN) specified by *integer*.

USING BACKUP CONTROLFILE — specifies that a backup control file is being used instead of the current control file.

TABLESPACE — recovers only the specified tablespaces. You can use this clause if the database is open or closed, provided the tablespaces to be recovered are offline.

DATAFILE — recovers the specified datafiles. You can use this clause when the database is open or closed, provided the datafiles to be recovered are offline.

STANDBY TABLESPACE | DATAFILE	reconstructs a lost or damaged datafile or tablespace in the standby database using archived redo log files copied from the primary database and a control file.
UNTIL [CONSISTENT WITH] CONTROLFILE	specifies that the recovery of an old standby datafile or tablespace uses the current standby database control file. However, any redo in advance of the standby controlfile will not be applied. The keywords CONSISTENT WITH are optional and are provided for semantic clarity.
LOGFILE	continues media recovery by applying the specified redo log file.
CONTINUE	continues multi-instance recovery after it has been interrupted to disable a thread.
CONTINUE DEFAULT	continues recovery using the redo log file that Oracle would automatically generate if no other logfile were specified. This clause is equivalent to specifying AUTOMATIC, except that Oracle does not prompt for a filename.
CANCEL	terminates cancel-based recovery.
managed_ recovery_clause	specifies sustained standby recovery mode. This mode assumes that the standby database is an active component of an overall standby database architecture. A primary database actively archives its redo log files to the standby site. As these archived redo logs arrive at the standby site, they become available for use by a managed standby recovery operation. Sustained standby recovery is restricted to media recovery. For more information on the parameters of this clause, see *Oracle8i Backup and Recovery Guide*. **Restrictions:** The same restrictions apply as are listed under *general_recovery_clause*.
TIMEOUT *integer*	specifies in minutes the wait period of the sustained recovery operation. The recovery process waits for *integer* minutes for a requested archived log redo to be available for writing to the standby database. If the redo log file does not become available within that time, the recovery process terminates with an error message. You can then issue the statement again to return to sustained standby recovery mode. If you do not specify this clause, the database remains in sustained standby recovery mode until you reissue the statement with the RECOVER CANCEL clause or until instance shutdown or failure.
CANCEL	terminates the sustained recovery operation after applying all the redo in the current archived redo file.
CANCEL IMMEDIATE	terminates the sustained recovery operation after applying all the redo in the current archived redo file or after the next redo log file read, whichever comes first. **Restriction**: This clause cannot be issued from the same session that issued the RECOVER MANAGED STANDBY DATABASE statement.
parallel_clause	specifies whether the recovery of media will be parallelized. For additional information, see the Notes to the *parallel_clause* of "CREATE TABLE" on page 7-359.
NOPARALLEL	specifies serial execution. This is the default.

	PARALLEL	causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.
	PARALLEL *integer*	specifies the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution processes. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

RENAME GLOBAL_NAME — changes the global name of the database. The *database* is the new database name and can be as long as eight bytes. The optional *domain* specifies where the database is effectively located in the network hierarchy. For more information on global names, see *Oracle8i Distributed Database Systems*.

Note: Renaming your database does not change global references to your database from existing database links, synonyms, and stored procedures and functions on remote databases. Changing such references is the responsibility of the administrator of the remote databases.

RENAME FILE — renames datafiles, tempfiles, or redo log file members. This clause renames only files in the control file. It does not actually rename them on your operating system. You must specify each filename using the conventions for filenames on your operating system before specifying this clause.

RESET COMPATIBILITY — marks the database to be reset to an earlier version of Oracle when the database is next restarted.

Note: RESET COMPATIBILITY works only if you have successfully disabled Oracle features that affect backward compatibility. For more information on downgrading to an earlier version of Oracle, see *Oracle8i Migration*.

You can use the following clauses only when your instance has the database open:

ENABLE THREAD — in a parallel server, enables the specified thread of redo log file groups. The thread must have at least two redo log file groups before you can enable it.

PUBLIC	makes the enabled thread available to any instance that does not explicitly request a specific thread with the initialization parameter THREAD. If you omit PUBLIC, the thread is available only to the instance that explicitly requests it with the initialization parameter THREAD.

DISABLE THREAD — disables the specified thread, making it unavailable to all instances. You cannot disable a thread if an instance using it has the database mounted.

CHARACTER SET
NATIONAL CHARACTER SET

CHARACTER SET changes the character set the database uses to store data. NATIONAL CHARACTER SET changes the national character set used to store data in columns specifically defined as NCHAR, NCLOB, or NVARCHAR2. Specify *character_set* without quotation marks.

WARNING: You cannot roll back an ALTER DATABASE CHARACTER SET or ALTER DATABASE NATIONAL CHARACTER SET statement. Therefore, you should perform a full backup before issuing either of these statements.

Restrictions:

- You must have SYSDBA system privilege, and you must start up the database in restricted mode (for example, with the SQL*Plus STARTUP RESTRICT command).
- The current character set must be a strict subset of the character set to which you change. That is, each character represented by a codepoint value in the source character set must be represented by the same codepoint value in the target character set. For a list of valid character sets, see *Oracle8i National Language Support Guide.*

datafile/tempfile_clauses

let you modify datafiles and tempfiles.

You can use any of the following clauses when your instance has the database mounted, open or closed, and the files involved are not in use:

CREATE DATAFILE

creates a new empty datafile in place of an old one. You can use this clause to re-create a datafile that was lost with no backup. The '*filename*' must identify a file that is or was once part of the database. The *filespec* specifies the name and size of the new datafile. If you omit the AS clause, Oracle creates the new file with the name and size as the file specified by '*filename*'.

During recovery, all archived redo logs written to since the original datafile was created must be applied to the new, empty version of the lost datafile.

Oracle creates the new file in the same state as the old file when it was created. You must perform media recovery on the new file to return it to the state of the old file at the time it was lost.

Restriction: You cannot create a new file based on the first datafile of the SYSTEM tablespace.

DATAFILE

affects your database files as follows:

ONLINE — brings the datafile online.

OFFLINE — takes the datafile offline. If the database is open, you must perform media recovery on the datafile before bringing it back online, because a checkpoint is not performed on the datafile before it is taken offline.

DROP takes a datafile offline when the database is in NOARCHIVELOG mode.

	RESIZE	attempts to increase or decrease the size of the datafile to the specified absolute size in bytes. Use K or M to specify this size in kilobytes or megabytes. There is no default, so you must specify a size. If sufficient disk space is not available for the increased size, or if the file contains data beyond the specified decreased size, Oracle returns an error.
	autoextend_ clause	enables or disables the automatic extension of a datafile. If you do not specify this clause, datafiles are not automatically extended. OFF disables autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER DATABASE AUTOEXTEND statements. ON enables autoextend. NEXT specifies in bytes the size of the next increment of disk space to be automatically allocated to the datafile when more extents are required. Use K or M to specify this size in kilobytes or megabytes. The default is one data block. MAXSIZE specifies the maximum disk space allowed for automatic extension of the datafile. UNLIMITED sets no limit on allocating disk space to the datafile.
	END BACKUP	avoids media recovery on database startup after an online tablespace backup was interrupted by a system failure or instance failure or SHUTDOWN ABORT.
	WARNING: Do not use ALTER TABLESPACE ... END BACKUP if you have restored any of the files affected from a backup. Media recovery is fully described in *Oracle8i Backup and Recovery Guide***.**	
TEMPFILE	Lets you resize your temporary datafile or specify the *autoextend_clause*, with the same effect as with a permanent datafile. **Restriction:** You cannot specify TEMPFILE unless the database is open.	
	DROP	drops *tempfile* from the database. The tablespace remains.
logfile_clauses	lets you add, drop, or modify log files.	
ARCHIVELOG	specifies that the contents of a redo log file group must be archived before the group can be reused. This mode prepares for the possibility of media recovery. Use this clause only after shutting down your instance normally or immediately with no errors and then restarting it, mounting the database in parallel server disabled mode.	
NOARCHIVELOG	specifies that the contents of a redo log file group need not be archived so that the group can be reused. This mode does not prepare for recovery after media failure.	

Use the ARCHIVELOG clause and NOARCHIVELOG clause only if your instance has the database mounted in parallel server disabled mode, but not open.

ADD LOGFILE — adds one or more redo log file groups to the specified thread, making them available to the instance assigned the thread.

- THREAD *integer* — is applicable only if you are using Oracle with the Parallel Server option in parallel mode. If you omit THREAD, the redo log file group is added to the thread assigned to your instance.
- GROUP *integer* — uniquely identifies the redo log file group among all groups in all threads and can range from 1 to the MAXLOGFILES value. You cannot add multiple redo log file groups having the same GROUP value. If you omit this parameter, Oracle generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance view V$LOG.
- *filespec* — Each *filespec* specifies a redo log file group containing one or more members, or copies. See the syntax description of *filespec* in "filespec" on page 7-490.

ADD LOGFILE MEMBER — adds new members to existing redo log file groups. Each new member is specified by '*filename*'. If the file already exists, it must be the same size as the other group members, and you must specify REUSE. If the file does not exist, Oracle creates a file of the correct size. You cannot add a member to a group if all of the group's members have been lost through media failure.

You can specify an existing redo log file group in one of these ways:

- GROUP *integer* — Specify the value of the GROUP parameter that identifies the redo log file group.
- *list of filenames* — List all members of the redo log file group. You must fully specify each filename according to the conventions of your operating system.

DROP LOGFILE — drops all members of a redo log file group. Specify a redo log file group as indicated for the ADD LOGFILE MEMBER clause.

- To drop the current log file group, you must first issue an ALTER SYSTEM SWITCH LOGFILE statement. See "ALTER SYSTEM" on page 7-95.
- You cannot drop a redo log file group if it needs archiving.
- You cannot drop a redo log file group if doing so would cause the redo thread to contain less than two redo log file groups.

DROP LOGFILE MEMBER — drops one or more redo log file members. Each 'filename' must fully specify a member using the conventions for filenames on your operating system.

- To drop a log file in the current log, you must first issue an ALTER SYSTEM SWITCH LOGFILE statement. See "ALTER SYSTEM" on page 7-95.
- You cannot use this clause to drop all members of a redo log file group that contains valid data. To perform this operation, use the DROP LOGFILE clause.

CLEAR LOGFILE — reinitializes an online redo log, optionally without archiving the redo log. CLEAR LOGFILE is similar to adding and dropping a redo log, except that the statement may be issued even if there are only two logs for the thread and also may be issued for the current redo log of a closed thread.

- UNARCHIVED — You must specify UNARCHIVED if you want to reuse a redo log that was not archived.

  WARNING: Specifying UNARCHIVED makes backups unusable if the redo log is needed for recovery.

Do not use CLEAR LOGFILE to clear a log needed for media recovery. If it is necessary to clear a log containing redo after the database checkpoint, you must first perform incomplete media recovery. The current redo log of an open thread can be cleared. The current log of a closed thread can be cleared by switching logs in the closed thread.

If the CLEAR LOGFILE statement is interrupted by a system or instance failure, then the database may hang. If this occurs, reissue the statement after the database is restarted. If the failure occurred because of I/O errors accessing one member of a log group, then that member can be dropped and other members added.

- UNRECOVERABLE DATAFILE — You must specify UNRECOVERABLE DATAFILE if you have taken the datafile offline with the database in ARCHIVELOG mode (that is, you specified ALTER DATABSE ... DATAFILE OFFLINE without the DROP keyword), and if the unarchived log to be cleared is needed to recover the datafile before bringing it back online. In this case, you must drop the datafile and the entire tablespace once the CLEAR LOGFILE statement completes.

controlfile_clauses

CREATE STANDBY CONTROLFILE — creates a control file to be used to maintain a standby database. For more information, see *Oracle8i Backup and Recovery Guide*. If the file already exists, you must specify REUSE.

BACKUP CONTROLFILE — backs up the current control file.

- TO '*filename*' — specifies the file to which the control file is backed up. You must fully specify the *filename* using the conventions for your operating system. If the specified file already exists, you must specify REUSE.
- TO TRACE — writes SQL statements to the database's trace file rather than making a physical backup of the control file. The SQL statements can start up the database, re-create the control file, and recover and open the database appropriately, based on the created control file.

  You can copy the statements from the trace file into a script file, edit the statements as necessary, and use the database if all copies of the control file are lost (or to change the size of the control file).

- RESETLOGS specifies that the SQL statement written to the trace file for starting the database is ALTER DATABASE OPEN RESETLOGS.
- NORESETLOGS specifies that the SQL statement written to the trace file for starting the database is ALTER DATABASE OPEN NORESETLOGS.

Examples

READ ONLY / READ WRITE Example The first statement below opens the database in read-only mode. The second statement returns the database to read-write mode and clears the online redo logs:

```
ALTER DATABASE OPEN READ ONLY;

ALTER DATABASE OPEN READ WRITE RESETLOGS;
```

PARALLEL Example The following statement performs tablespace recovery using parallel recovery processes:

```
ALTER DATABASE
   RECOVER TABLESPACE binky
   PARALLEL;
```

Redo Log File Group Example The following statement adds a redo log file group with two members and identifies it with a GROUP parameter value of 3:

```
ALTER DATABASE stocks
  ADD LOGFILE GROUP 3
    ('diska:log3.log' ,
     'diskb:log3.log') SIZE 50K;
```

Redo Log File Group Member Example The following statement adds a member to the redo log file group added in the previous example:

```
ALTER DATABASE stocks
   ADD LOGFILE MEMBER 'diskc:log3.log'
   TO GROUP 3;
```

Dropping a Log File Member The following statement drops the redo log file member added in the previous example:

```
ALTER DATABASE stocks
    DROP LOGFILE MEMBER 'diskc:log3.log';
```

Renaming a Log File Member The following statement renames a redo log file member:

```
ALTER DATABASE stocks
    RENAME FILE 'diskb:log3.log' TO 'diskd:log3.log';
```

The above statement only changes the member of the redo log group from one file to another. The statement does not actually change the name of the file 'DISKB:LOG3.LOG' to 'DISKD:LOG3.LOG'. You must perform this operation through your operating system.

Dropping All Log File Group Members The following statement drops all members of the redo log file group 3:

```
ALTER DATABASE stocks DROP LOGFILE GROUP 3;
```

Adding a Redo Log File Group The following statement adds a redo log file group containing three members to thread 5 (in an Oracle Parallel Server environment) and assigns it a GROUP parameter value of 4:

```
ALTER DATABASE stocks
    ADD LOGFILE THREAD 5 GROUP 4
        ('diska:log4.log',
        'diskb:log4:log',
        'diskc:log4.log' );
```

Disabling a Parallel Server Thread The following statement disables thread 5 in a parallel server:

```
ALTER DATABASE stocks
    DISABLE THREAD 5;
```

Enabling a Parallel Server Thread The following statement enables thread 5 in a parallel server, making it available to any Oracle instance that does not explicitly request a specific thread:

```
ALTER DATABASE stocks
    ENABLE PUBLIC THREAD 5;
```

Creating a New Datafile The following statement creates a new datafile 'DISK2:DB1.DAT' based on the file 'DISK1:DB1.DAT':

```
ALTER DATABASE
    CREATE DATAFILE 'disk1:db1.dat' AS 'disk2:db1.dat';
```

Changing the Global Database Name The following statement changes the global name of the database and includes both the database name and domain:

```
ALTER DATABASE
    RENAME GLOBAL_NAME TO sales.australia.acme.com;
```

Character Set Example The following statements change the database character set and national character set to the WE8ISO8859P1 character set:

```
ALTER DATABASE db1 CHARACTER SET WE8ISO8859P1;
ALTER DATABASE db1 NATIONAL CHARACTER SET WE8ISO8859P1;
```

The database name is optional, and the character set name is specified without quotation marks.

Resizing a Datafile The following statement attempts to change the size of datafile 'DISK1:DB1.DAT':

```
ALTER DATABASE
    DATAFILE 'disk1:db1.dat' RESIZE 10 M;
```

Clearing a Log File The following statement clears a log file:

```
ALTER DATABASE
    CLEAR LOGFILE 'disk3:log.dbf';
```

Database Recovery Examples The following statement performs complete recovery of the entire database, letting Oracle generate the name of the next archived redo log file needed:

```
ALTER DATABASE
  RECOVER AUTOMATIC DATABASE;
```

The following statement explicitly names a redo log file for Oracle to apply:

```
ALTER DATABASE
    RECOVER LOGFILE 'diska:arch0006.arc';
```

The following statement performs time-based recovery of the database:

```
ALTER DATABASE
    RECOVER AUTOMATIC UNTIL TIME '1998-10-27:14:00:00';
```

Oracle recovers the database until 2:00 pm on October 27, 1998.

The following statement recovers the tablespace USER5:

```
ALTER DATABASE
    RECOVER TABLESPACE user5;
```

The following statement recovers the standby datafile /FINANCE/STBS_21.f, using the corresponding datafile in the original standby database, plus all relevant archived logs and the current standby database control file:

```
ALTER DATABASE
   RECOVER STANDBY DATAFILE '/finance/stbs_21.f'
   UNTIL CONTROLFILE;
```

Managed Standby Database Examples The following statement recovers the standby database in managed (sustained) standby recovery mode:

```
ALTER DATABASE
   RECOVER MANAGED STANDBY DATABASE;
```

The following statement puts the database in managed standby recovery mode. The sustained recovery process will wait up to 60 minutes for the next archive log:

```
ALTER DATABASE
   RECOVER MANAGED STANDBY DATABASE TIMEOUT 60;
```

If each subsequent log arrives within 60 minutes of the last log, sustained recovery continues indefinitely or until manually terminated.

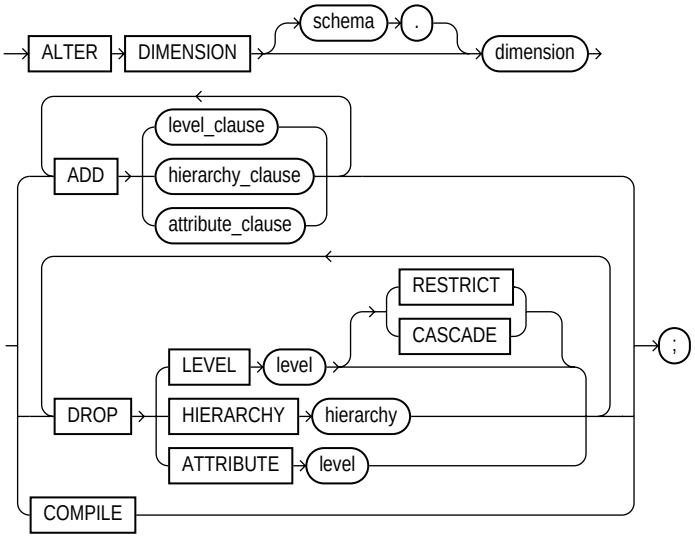
The following statement terminates the managed recovery operation:

```
ALTER DATABASE
   RECOVER MANAGED STANDBY DATABASE CANCEL IMMEDIATE;
```

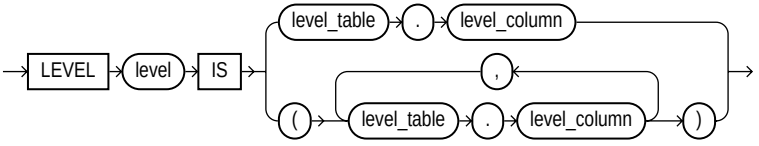
The sustained recovery operation terminates before the next group of redo is read from the current redo log file. Media recovery ends in the "middle" of applying redo from the current redo log file.

ALTER DIMENSION

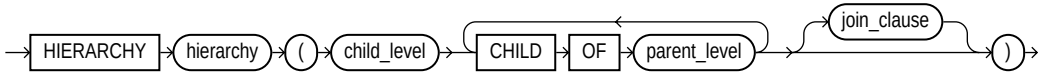
Syntax



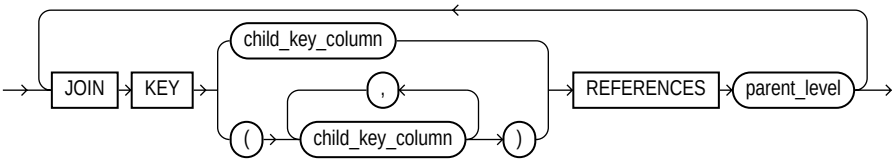
level_clause::=



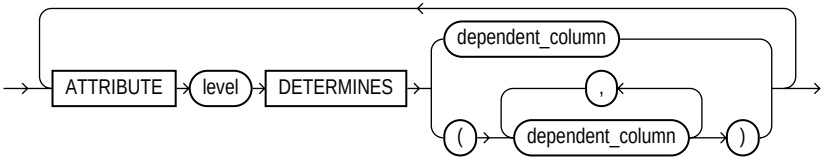
hierarchy_clause::=



join_clause::=



attribute_clause::=



Purpose

To change the hierarchical relationships or dimension attributes of a dimension. For more information on dimensions, see "CREATE DIMENSION" on page 7-259.

Prerequisites

The dimension must be in your schema or you must have the ALTER ANY DIMENSION system privilege to use this statement.

A dimension is always altered under the rights of the owner.

Keywords and Parameters

The following keywords and parameters have meaning unique to ALTER DIMENSION. The remaining keywords and parameters have the same functionality that they have in the CREATE DIMENSION statement. See "CREATE DIMENSION" on page 7-259.

schema	is the schema of the dimension you want to modify. If you do not specify *schema*, Oracle assumes the dimension is in your own schema.
dimension	is the name of the dimension. This dimension must already exist.

ADD	lets you add a level, hierarchy, or attribute to the dimension. Adding one of these elements does not invalidate any existing materialized view. Oracle processes ADD LEVEL clauses prior to any other ADD clauses.
DROP	lets you drop a level, hierarchy, or attribute from the dimension. Any level, hierarchy, or attribute you specify must already exist. **Restriction:** If any attributes or hierarchies reference a level, you cannot drop the level until you either drop all the referencing attributes and hierarchies or specify CASCADE. CASCADE — causes Oracle to drop any attributes or hierarchies that reference the level, along with the level itself. RESTRICT — prevents Oracle from dropping a level that is referenced by any attributes or hierarchies. This is the default.
COMPILE	explicitly recompiles an invalidated dimension. Oracle automatically compiles a dimension when you issue an ADD clause or DROP clause. However, if you alter an object referenced by the dimension (for example, if you drop and then re-create a table referenced in the dimension), the dimension will be invalidated, and you must recompile it explicitly.

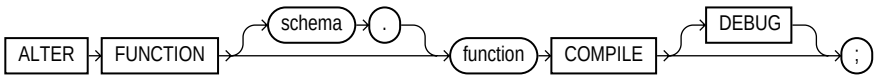
Examples

This example modifies the TIME dimension:

```
ALTER DIMENSION time
   DROP HIERARCHY week_month;
ALTER DIMENSION time
   DROP ATTRIBUTE cur_date;
ALTER DIMENSION time
   ADD LEVEL day IS time_tab.t_day
   ADD ATTRIBUTE day DETERMINES t_holiday;
```

ALTER FUNCTION

Syntax



Purpose

To recompile an invalid standalone stored function. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

The ALTER FUNCTION statement is similar to "ALTER PROCEDURE" on page 7-62. For information on how Oracle recompiles functions and procedures, see *Oracle8i Concepts*.

> **Note:** This statement does not change the declaration or definition of an existing function. To redeclare or redefine a function, use the CREATE FUNCTION statement with the OR REPLACE clause; see "CREATE FUNCTION" on page 7-266.

Prerequisites

The function must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Keywords and Parameters

schema	is the schema containing the function. If you omit *schema*, Oracle assumes the function is in your own schema.
function	is the name of the function to be recompiled.
COMPILE	causes Oracle to recompile the function. The COMPILE keyword is required. If Oracle does not compile the function successfully, you can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.
DEBUG	instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.

Example

To explicitly recompile the function GET_BAL owned by the user MERRIWEATHER, issue the following statement:

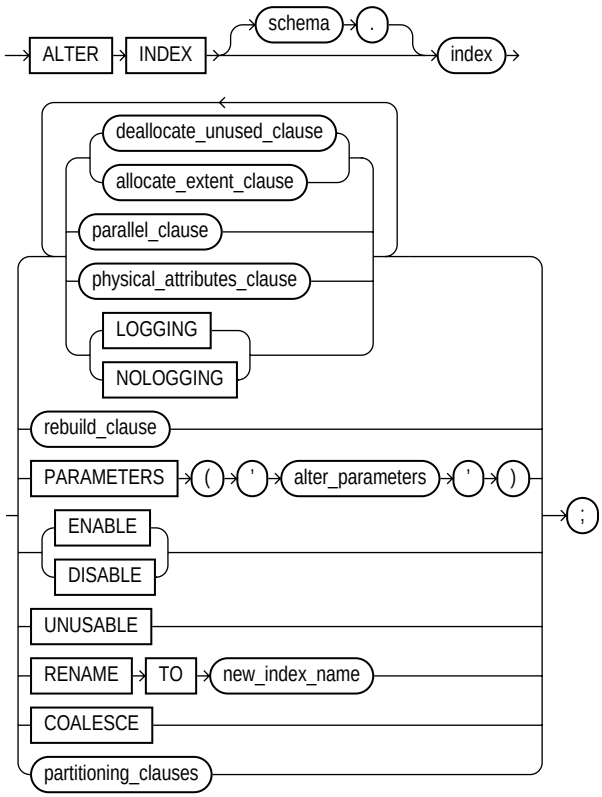
```
ALTER FUNCTION merriweather.get_bal
   COMPILE;
```

If Oracle encounters no compilation errors while recompiling GET_BAL, GET_BAL becomes valid. Oracle can subsequently execute it without recompiling it at run time. If recompiling GET_BAL results in compilation errors, Oracle returns an error, and GET_BAL remains invalid.

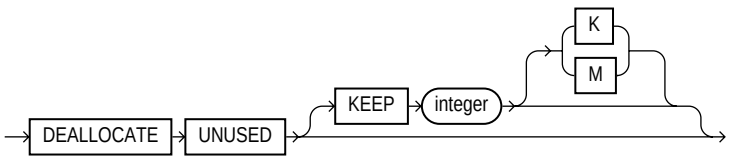
Oracle also invalidates all objects that depend upon GET_BAL. If you subsequently reference one of these objects without explicitly recompiling it first, Oracle recompiles it implicitly at run time.

ALTER INDEX

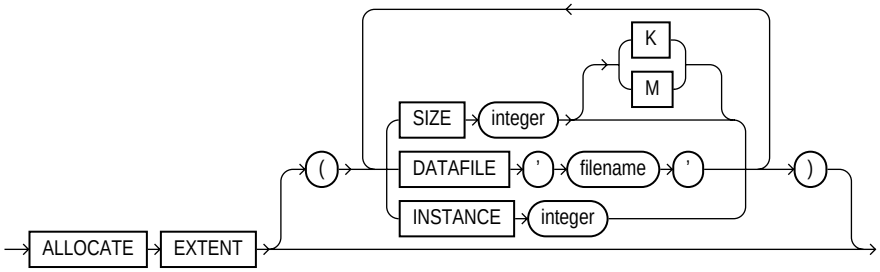
Syntax



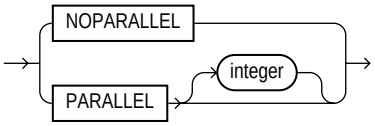
deallocate_unused_clause::=



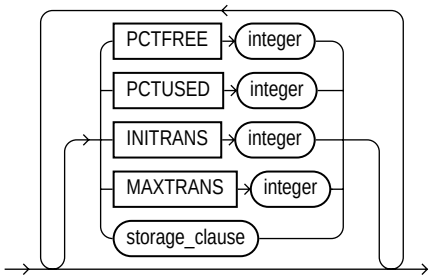
allocate_extent_clause::=



parallel_clause::=

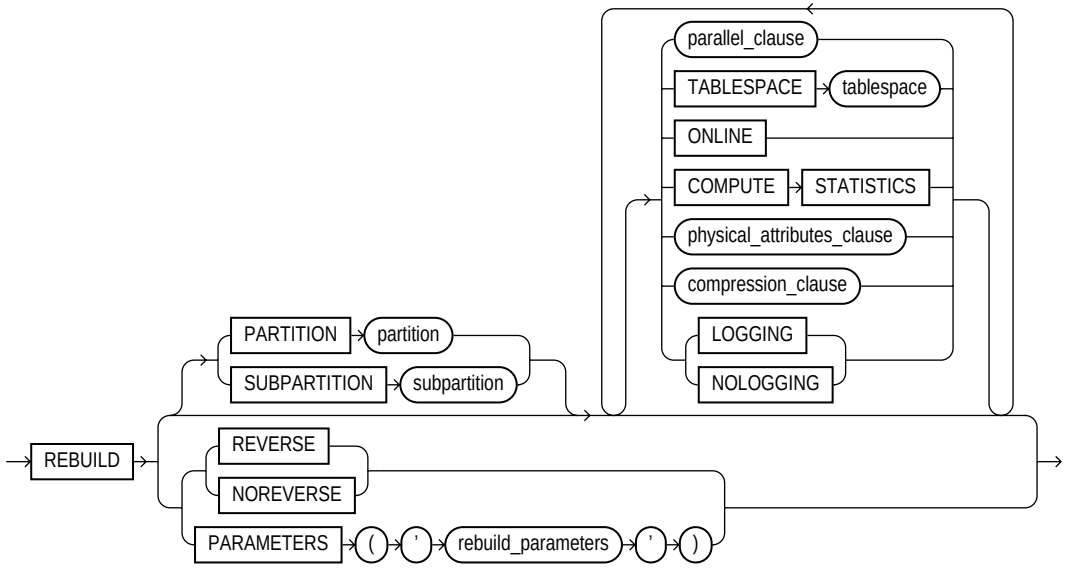


physical_attributes_clause::=

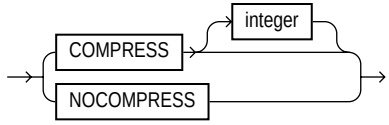


storage_clause: See "storage_clause" on page 7-575.

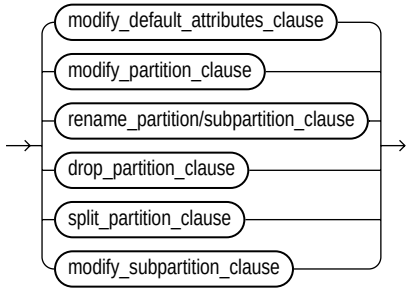
rebuild_clause::=



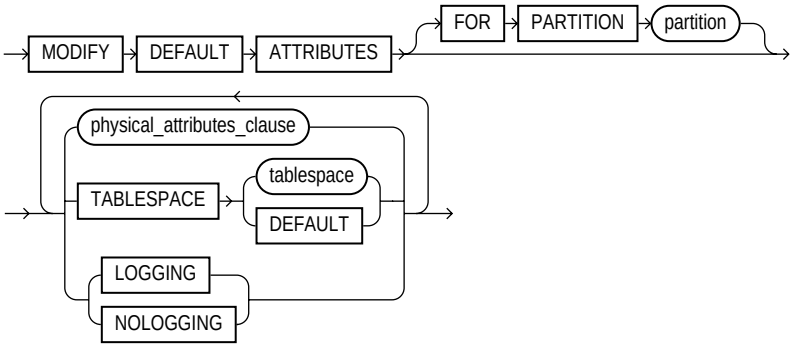
compression_clause::=



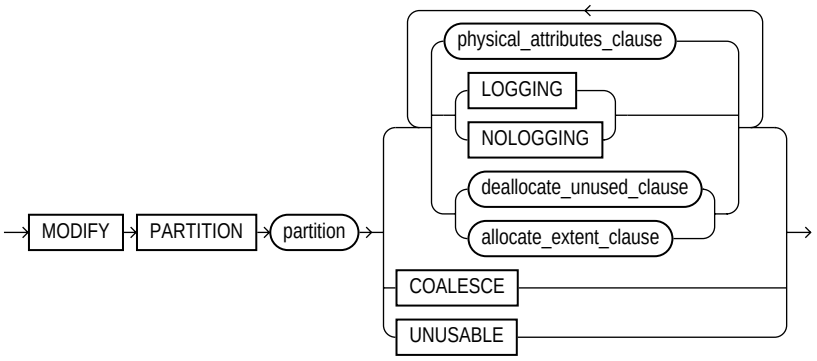
partitioning_clauses::=



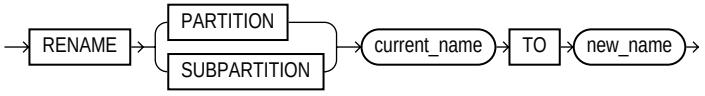
modify_default_attributes_clause::=



modify_partition_clause::=

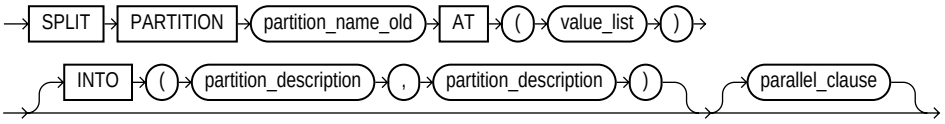


rename_partition/ subpartition_clause::=

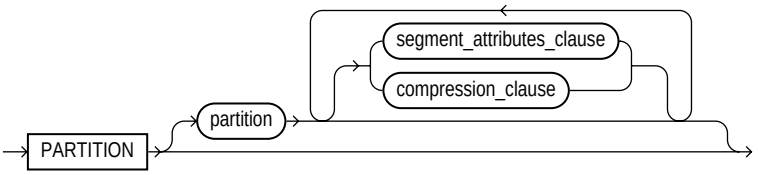


drop_partition_clause::=

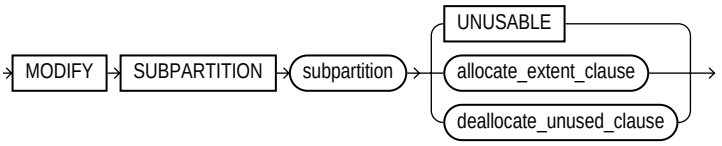
split_partition_clause::=



partition_description::=



modify_subpartition_clause::=



Purpose

To change or rebuild an existing index.

For information on creating an index, see "CREATE INDEX" on page 7-273.

Prerequisites

The index must be in your own schema or you must have ALTER ANY INDEX system privilege.

Schema object privileges are granted on the parent index, not on individual index partitions or subpartitions.

You must have tablespace quota to modify, rebuild, or split an index partition or to modify or rebuild an index subpartition.

Keywords and Parameters

schema is the schema containing the index. If you omit *schema*, Oracle assumes the index is in your own schema.

index is the name of the index to be altered.

Restrictions:

- If *index* is a domain index, you can specify only the PARAMETERS clause, the RENAME clause, or the *rebuild_clause* (with or without the PARAMETERS clause). No other clauses are valid.
- You cannot alter or rename a domain index that is marked LOADING or FAILED. If an index is marked FAILED, the only clause you can specify is REBUILD. For information on the LOADING and FAILED states of domain indexes, see *Oracle8i Data Cartridge Developer's Guide.*

deallocate_unused_clause explicitly deallocates unused space at the end of the index and makes the freed space available for other segments in the tablespace. Only unused space above the high water mark can be freed. For more information on this clause, see "ALTER TABLE" on page 7-113.

If *index* is range-partitioned or hash-partitioned, Oracle deallocates unused space from each index partition. If *index* is a local index on a composite-partitioned table, Oracle deallocates unused space from each index subpartition.

Restrictions:

- You cannot specify this clause for an index on a temporary table.
- You cannot specify this clause and also specify the *rebuild_clause*.

KEEP specifies the number of bytes above the high water mark that the index will have after deallocation. If the number of remaining extents are less than MINEXTENTS, then MINEXTENTS is set to the current number of extents. If the initial extent becomes smaller than INITIAL, then INITIAL is set to the value of the current initial extent. If you omit KEEP, all unused space is freed.

For a complete description of this clause, see "ALTER TABLE" on page 7-113.

allocate_extent_clause explicitly allocates a new extent for the index. For a local index on a hash-partitioned table, Oracle allocates a new extent for each partition of the index.

Restriction: You cannot specify this clause for an index on a temporary table or for a range-partitioned or composite-partitioned index.

SIZE specifies the size of the extent in bytes. Use K or M to specify the extent size in kilobytes or megabytes. If you omit SIZE, Oracle determines the size based on the values of the index's storage parameters.

DATAFILE — specifies one of the datafiles in the index's tablespace to contain the new extent. If you omit DATAFILE, Oracle chooses the datafile.

INSTANCE — makes the new extent available to the specified instance. An instance is identified by the value of its initialization parameter INSTANCE_NUMBER. If you omit this parameter, the extent is available to all instances. Use this parameter only if you are using Oracle with the Parallel Server option in parallel mode.

Explicitly allocating an extent with this clause does not change the values of the NEXT and PCTINCREASE storage parameters, so does not affect the size of the next extent to be allocated.

parallel_clause — changes the default degree of parallelism for queries and DML on the index. For additional information, see the Notes to the *parallel_clause* of "CREATE TABLE" on page 7-359.

NOPARALLEL — specifies serial execution. This is the default.

PARALLEL — causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances multiplied by the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — specifies the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution processes. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Restriction: You cannot specify this clause for an index on a temporary table.

physical_attributes_clause — lets you change the values of parameters for a nonpartitioned index, all partitions and subpartitions of a partitioned index, a specified partition, or all subpartitions of a specified partition. See these parameters in "CREATE TABLE" on page 7-359.

Restrictions:

- You cannot specify this clause for an index on a temporary table.
- You cannot specify the PCTUSED parameter when altering an index.
- You cannot change the value of the PCTFREE parameter for the index as a whole (ALTER INDEX) or for a partition (ALTER INDEX ... MODIFY PARTITION). You can specify PCTFREE in all other forms of the ALTER INDEX statement.

storage_clause — changes the storage parameters for a nonpartitioned index, index partition, or all partitions of a partitioned index, or default values of these parameters for a partitioned index. See the "storage_clause" on page 7-575.

LOGGING | NOLOGGING

LOGGING | NOLOGGING specifies that subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against a nonpartitioned index, a range or hash index partition, or all partitions or subpartitions of a composite-partitioned index will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file.

In NOLOGGING mode, data is modified with minimal logging (to mark new extents invalid and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not logged. Therefore, if you cannot afford to lose this index, you must take a backup after the operation in NOLOGGING mode.

If the database is run in ARCHIVELOG mode, media recovery from a backup taken before an operation in LOGGING mode will re-create the index. However, media recovery from a backup taken before an operation in NOLOGGING mode will not re-create the index.

An index segment can have logging attributes different from those of the base table and different from those of other index segments for the same base table.

Restriction: You cannot specify this clause for an index on a temporary table.

For more information about LOGGING and parallel DML, see *Oracle8i Concepts* and the *Oracle8i Parallel Server Concepts and Administration*.

RECOVERABLE | UNRECOVERABLE

These keywords are deprecated and have been replaced with LOGGING and NOLOGGING, respectively. Although RECOVERABLE and UNRECOVERABLE are supported for backward compatibility, Oracle Corporation strongly recommends that you use the LOGGING and NOLOGGING keywords.

RECOVERBLE is not a valid keyword for creating partitioned tables or LOB storage characteristics. UNRECOVERABLE is not a valid keyword for creating partitioned or index-organized tables. Also, it can be specified only with the AS subquery clause of CREATE INDEX.

rebuild_clause

re-creates an existing index or one of its partitions or subpartitions. For a function-based index, this clause also enables the index. If the function on which the index is based does not exist, the rebuild statement will fail.

Restrictions:

- You cannot rebuild an index on a temporary table.
- You cannot rebuild an entire partitioned index. You must rebuild each partition or subpartition, as described below.
- You cannot also specify the *deallocate_unused_clause* in this statement.
- You cannot change the value of the PCTFREE parameter for the index as a whole (ALTER INDEX) or for a partition (ALTER INDEX ... MODIFY PARTITION). You can specify PCTFREE in all other forms of the ALTER INDEX statement.

PARTITION *partition*	rebuilds one partition of an index. You can also use this clause to move an index partition to another tablespace or to change a create-time physical attribute. For more information about partition maintenance operations, see the *Oracle8i Administrator's Guide*. **Restriction:** You cannot specify this clause for a local index on a composite-partitioned table. Instead, use the REBUILD SUBPARTITION clause.
SUBPARTITION *subpartition*	rebuilds one subpartition of an index. You can also use this clause to move an index subpartition to another tablespace. If you do not specify TABLESPACE, the subpartition is rebuilt in the same tablespace. **Restrictions:** The only parameters you can specify for a subpartition are TABLESPACE and the *parallel_clause*.
REVERSE | NOREVERSE	specifies whether the bytes of the index block are stored in reverse order. ■ REVERSE stores the bytes of the index block in reverse order and excludes the rowid when the index is rebuilt. ■ NOREVERSE stores the bytes of the index block without reversing the order when the index is rebuilt. Rebuilding a REVERSE index without the NOREVERSE keyword produces a rebuilt, reverse-keyed index. **Restrictions:** ■ You cannot reverse a bitmap index or an index-organized table. ■ You cannot specify REVERSE or NOREVERSE for a partition or subpartition.
TABLESPACE	specifies the tablespace where the rebuilt index, index partition, or index subpartition will be stored. The default is the default tablespace where the index or partition resided before you rebuilt it.
COMPRESS	enables key compression, which eliminates repeated occurrence of key column values. Use *integer* to specify the prefix length (number of prefix columns to compress). ■ For unique indexes, the range of valid prefix length values is from 1 to the number of key columns minus 1. The default prefix length is the number of key columns minus 1. ■ For nonunique indexes, the range of valid prefix length values is from 1 to the number of key columns. The default prefix length is number of key columns. Oracle compresses only nonpartitioned indexes that are nonunique or unique indexes of at least two columns. **Restriction**: You cannot specify COMPRESS for a bitmapped index.

	NOCOMPRESS	disables key compression. This is the default.
	ONLINE	specifies that DML operations on the table or partition are allowed during rebuilding of the index. **Restriction**: Parallel DML is not supported during online index building. If you specify ONLINE and then issue parallel DML statements, Oracle returns an error.
	COMPUTE STATISTICS	enables you to collect statistics at relatively little cost during the rebuilding of an index. These statistics are stored in the data dictionary for ongoing use by the optimizer in choosing a plan of execution for SQL statements. The types of statistics collected depend on the type of index you are rebuilding. **Note**: If you create an index using another index (instead of a table), the original index might not provide adequate statistical information. Therefore, Oracle generally uses the base table to compute the statistics, which will improve the statistics but may negatively affect performance.
	.	Additional methods of collecting statistics are available in PL/SQL packages and procedures. See *Oracle8i Supplied Packages Reference.*
	LOGGING | NOLOGGING	specifies whether the `ALTER INDEX...REBUILD` operation will be logged.

PARAMETERS

applies only to **domain indexes**. This clause specifies the parameter string for altering the index (or, in the *rebuild_clause*, rebuilding the index). The maximum length of the parameter string is 1000 characters. This string is passed uninterpreted to the appropriate indextype routine. For more information on these routines, see *Oracle8i Data Cartridge Developer's Guide*. For more information on domain indexes, see "CREATE INDEX" on page 7-273.

Restrictions:

- You cannot specify this clause for any indexes other than domain indexes.
- The parameter string is passed to the appropriate routine only if *index* is not marked UNUSABLE.

ENABLE

applies only to a **function-based index** that has been disabled because a user-defined function used by the index was dropped or replaced. This clause enables such an index if

- the function is currently valid,
- the signature of the current function matches the signature of the function when the index was created, and
- the function is currently marked as DETERMINISTIC.

Restriction: You cannot specify any other clauses of ALTER INDEX in the same statement with ENABLE.

DISABLE — applies only to a **function-based index**. This clause enables you to disable the use of a function-based index. You might want to do so, for example, while working on the body of the function. Afterward you can either rebuild the index or specify another ALTER INDEX statement with the ENABLE keyword.

UNUSABLE — marks the index or index partition(s) or index subpartition(s) UNUSABLE. An unusable index must be rebuilt, or dropped and re-created, before it can be used. While one partition is marked UNUSABLE, the other partitions of the index are still valid. You can execute statements that require the index if the statements do not access the unusable partition. You can also split or rename the unusable partition before rebuilding it.

Restriction: You cannot specify this clause for an index on a temporary table.

RENAME TO — renames *index* to *new_index_name*. The *new_index_name* is a single identifier and does not include the schema name.

COALESCE — instructs Oracle to merge the contents of index blocks where possible to free blocks for reuse. For more information on space management and coalescing indexes, see *Oracle8i Administrator's Guide*.

Restriction: You cannot specify this clause for an index on a temporary table.

partitioning_clauses: The remainder of the clauses of the ALTER INDEX statement are valid only for partitioned indexes.

Restrictions:

- You cannot specify any of these clauses for an index on a temporary table.
- You can combine several operations on the base index into one ALTER INDEX statement (except RENAME and REBUILD), but you cannot combine partition operations with other partition operations or with operations on the base index.

modify_default_attributes_clause — specifies new values for the default attributes of a partitioned index.

Restriction: The only attribute you can specify for an index on a hash-partitioned or composite-partitioned table is TABLESPACE.

TABLESPACE — specifies the default tablespace for new partitions of an index or subpartitions of an index partition.

LOGGING | NOLOGGING — specifies the default logging attribute of a partitioned index or an index partition.

FOR PARTITION *partition* — specifies the default attributes for the subpartitions of a partition of a local index on a composite-partitioned table.

modify_partition_clause — modifies the real physical attributes, logging attribute, or storage characteristics of index partition *partition* or its subpartitions.

Restriction: You cannot specify the *physical_attributes_clause* for an index on a hash-partitioned table.

	Note: If the index is a local index on a composite-partitioned table, the changes you specify here will override any attributes specified earlier for the subpartitions of index, as well as establish default values of attributes for future subpartitions of that partition. To change the default attributes of the partition without overriding the attributes of subpartitions, use ALTER TABLE ... MODIFY DEFAULT ATTRIBUTES OF PARTITION.
rename_partition/ subpartition_clause	renames index partition or subpartition to *new_name*.
drop_partition_ clause	removes a partition and the data in it from a partitioned global index. When you drop a partition of a global index, Oracle marks the index's next partition UNUSABLE. You cannot drop the highest partition of a global index.
split_partition_ clause	splits a partition of a global partitioned index into two partitions, adding a new partition to the index. Splitting a partition marked UNUSABLE results in two partitions, both marked UNUSABLE. You must rebuild the partitions before you can use them. Splitting a usable partition results in two partitions populated with index data. Both new partitions are usable. AT (*value_list*) — specifies the new noninclusive upper bound for *split_partition_1*. The *value_list* must evaluate to less than the presplit partition bound for *partition_name_old* and greater than the partition bound for the next lowest partition (if there is one). INTO — describes the two partitions resulting from the split.
partition_ description	specifies (optionally) the name and physical attributes of each of two partitions resulting from a split.
modify_ subpartition_clause	lets you mark UNUSABLE or allocate or deallocate storage for a subpartition of a local index on a composite-partitioned table. All other attributes of such a subpartition are inherited from partition-level default attributes.

Examples

Modifying Real Attributes This statement alters SCOTT's CUSTOMER index so that future data blocks within this index use 5 initial transaction entries and an incremental extent of 100 kilobytes:

```
ALTER INDEX scott.customer
    INITRANS 5
    STORAGE (NEXT 100K);
```

If the SCOTT.CUSTOMER index is partitioned, this statement also alters the default attributes of future partitions of the index. New partitions added in the future will use 5 initial transaction entries and an incremental extent of 100K.

Dropping an Index Partition The following statement drops index partition IX_ANTARTICA:

```
ALTER INDEX sales_area_ix
  DROP PARTITION ix_antarctica;
```

Modifying Default Attributes This statement alters the default attributes of local partitioned index SALES_IX3. New partitions added in the future will use 5 initial transaction entries and an incremental extent of 100K:

```
ALTER INDEX sales_ix3
  MODIFY DEFAULT ATTRIBUTES INITRANS 5 STORAGE ( NEXT 100K );
```

Marking an Index Unusable The following statement marks the IDX_ACCTNO index as UNUSABLE:

```
ALTER INDEX idx_acctno UNUSABLE;
```

Marking a Partition Unusable The following statement marks partition IDX_FEB96 of index IDX_ACCTNO as UNUSABLE:

```
ALTER INDEX idx_acctno MODIFY PARTITION idx_feb96 UNUSABLE;
```

Changing MAXEXTENTS The following statement changes the maximum number of extents for partition BRIX_NY and changes the logging attribute:

```
ALTER INDEX branch_ix MODIFY PARTITION brix_ny
  STORAGE( MAXEXTENTS 30 ) LOGGING;
```

Disabling Parallel Queries The following statement sets the parallel attributes for index ARTIST_IX so that scans on the index will not be parallelized:

```
ALTER INDEX artist_ix NOPARALLEL;
```

Rebuilding a Partition The following statement rebuilds partition P063 in index ARTIST_IX. The rebuilding of the index partition will not be logged:

```
ALTER INDEX artist_ix
  REBUILD PARTITION p063 NOLOGGING;
```

Renaming an Index The following statement renames an index:

```
ALTER INDEX emp_ix1 RENAME TO employee_ix1;
```

Renaming an Index Partition The following statement renames an index partition:

```
ALTER INDEX employee_ix1 RENAME PARTITION emp_ix1_p3
  TO employee_ix1_p3;
```

Splitting a Partition The following statement splits partition PARTNUM_IX_P6 in partitioned index PARTNUM_IX into PARTNUM_IX_P5 and PARTNUM_IX_P6:

```
ALTER INDEX partnum_ix
  SPLIT PARTITION partnum_ix_p6 AT ( 5001 )
  INTO ( PARTITION partnum_ix_p5 TABLESPACE ts017 LOGGING,
         PARTITION partnum_ix_p6 TABLESPACE ts004 );
```

The second partition retains the name of the old partition.

Storing Index Blocks in Reverse Order The following statement rebuilds index EMP_IX so that the bytes of the index block are stored in REVERSE order:

```
ALTER INDEX emp_ix REBUILD REVERSE;
```

Collecting Index Statistics The following statement collects statistics on the nonpartitioned EMP_INDX index:

```
ALTER INDEX emp_indx REBUILD COMPUTE STATISTICS;
```

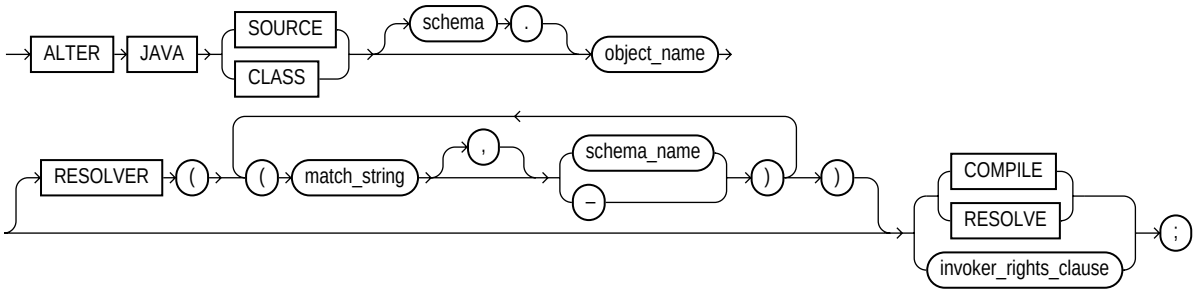
The type of statistics collected depends on the type of index you are rebuilding. For more information, refer to *Oracle8i Concepts*.

PARALLEL Example The following statement causes the index to be rebuilt from the existing index by using parallel parallel execution processes to scan the old and to build the new index:

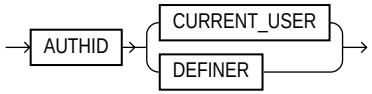
```
ALTER INDEX emp_idx
   REBUILD
   PARALLEL;
```

ALTER JAVA

Syntax



invoker_rights_clause::=



Purpose

To force the resolution of a Java class schema object or compilation of a Java source schema object. (You cannot call the methods of a Java class before all its external references to Java names are associated with other classes.)

For more information on resolving Java classes and compiling Java sources, see *Oracle8i Java Stored Procedures Developer's Guide.*

Prerequisites

The Java source or class must be in your own schema, or you must have the ALTER ANY PROCEDURE system privilege. You must also have the EXECUTE object privilege on Java classes.

Keywords and Parameters

JAVA SOURCE	compiles a Java source schema object.

JAVA CLASS	resolves a Java class schema object.
object_name	specifies a previously created Java class or source schema object.
RESOLVER	specifies how schemas are searched for referenced fully specified Java names, using the mapping pairs specified when the Java class or source was created. For more information, see "CREATE JAVA" on page 7-293.
RESOLVE | COMPILE	are synonymous keywords. They specify that Oracle should attempt to resolve the primary Java class schema object. ■ When applied to a class, resolution of referenced names to other class schema objects occurs. ■ When applied to a source, source compilation occurs.
invoker_rights_clause	specifies whether the methods of the class execute with the privileges and in the schema of the user who defined it or with the privileges and in the schema of CURRENT_USER. For information on how CURRENT_USER is determined, see *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals*. This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type. For more information refer to *Oracle8i Java Stored Procedures Developer's Guide*. AUTHID CURRENT_USER — specifies that the methods of the class execute with the privileges of CURRENT_USER. This clause is the default and creates an "invoker-rights class." This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the methods reside. AUTHID DEFINER — specifies that the methods of the class execute with the privileges of the user who defined it. This clause also specifies that external names resolve in the schema where the methods reside.

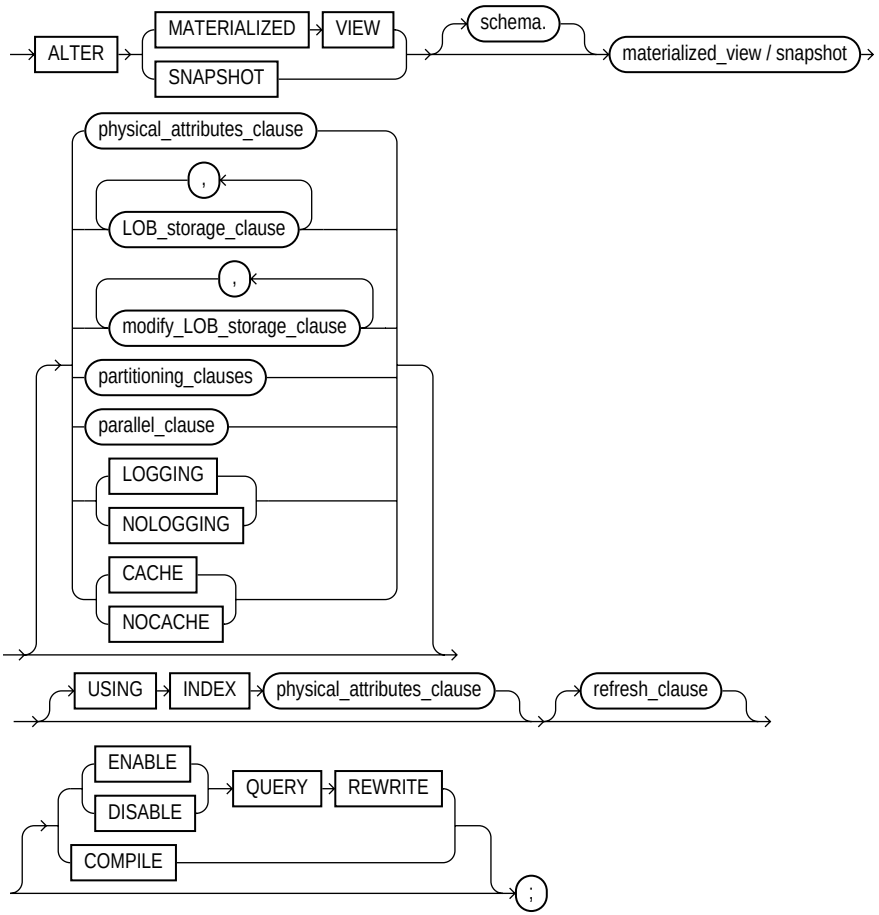
Example

The following statement forces the resolution of a Java class:

```
ALTER JAVA CLASS "Agent"
   RESOLVER (("/home/java/bin/*" scott)(* public))
   RESOLVE;
```

ALTER MATERIALIZED VIEW / SNAPSHOT

Syntax

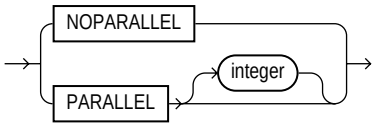


LOB_storage_clause: See "ALTER TABLE" on page 7-113.

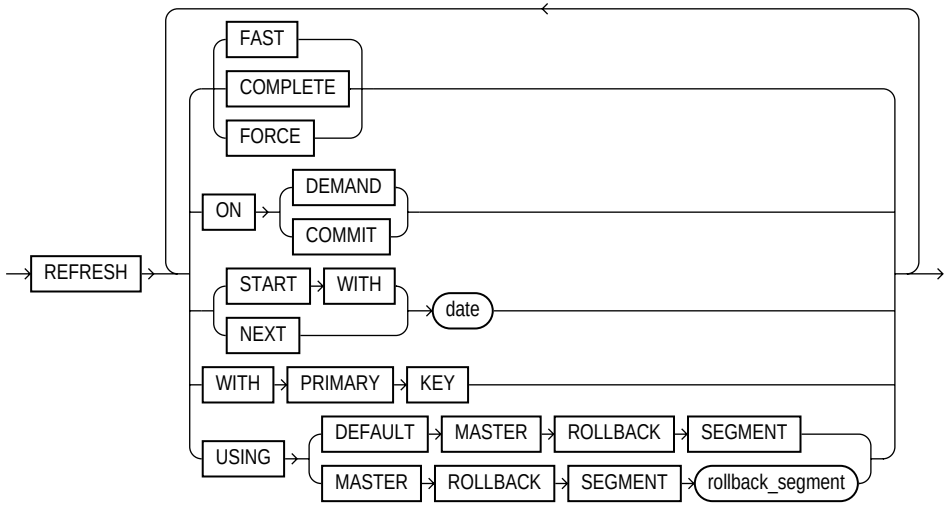
modify_LOB_storage_clause: See "ALTER TABLE" on page 7-113.

partitioning_clauses: See "ALTER TABLE" on page 7-113.

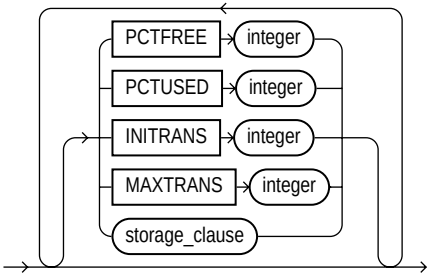
parallel_clause::=



refresh_clause::=



physical_attributes_clause::=



storage_clause: See the "storage_clause" on page 7-575.

Purpose

To change the storage characteristics, refresh mode or time, or type of an existing materialized view.

To enable or disable query rewrite.

The terms "snapshot" and "materialized view" are synonymous. Both refer to a table that contains the results of a query of one or more tables, each of which may be located on the same or on a remote database.

Replication and warehouse environments sometimes use different terms to describe the same thing. In this reference, **master tables** (a replication term) and **detail tables** (a warehouse term) both refer to the tables referenced by a materialized view.

For more information on materialized views, including a brief description of the different types of materialized views, see "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 7-300. For information on materialized views in a replication environment, see *Oracle8i Replication*. For information on materialized views in a data warehousing environment, see *Oracle8i Tuning*.

Prerequisites

To alter a materialized view's storage parameters, the materialized view must be contained in your own schema, or you must have the ALTER ANY SNAPSHOT or ALTER ANY MATERIALIZED VIEW system privilege.

To enable a materialized view for query rewrite:

- If all the master tables in the materialized view are in your schema, you must have the QUERY REWRITE privilege.
- If any of the master tables are in another schema, you must have the GLOBAL QUERY REWRITE privilege.
- If the materialized view is in another user's schema, *both you and the owner of that schema* must have the appropriate QUERY REWRITE privilege described in the preceding two items.

For detailed information about the prerequisites for ALTER MATERIALIZED VIEW, see *Oracle8i Replication*.

Keywords and Parameters

schema is the schema containing the materialized view. If you omit *schema*, Oracle assumes the materialized view is in your own schema.

materialized view / snapshot is the name of the materialized view to be altered.

physical_attributes_clause change the values of the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and the storage characteristics for the internal table that Oracle uses to maintain the materialized view's data. For more information, see "ALTER TABLE" on page 7-113 and the *"storage_clause" on page 7-575*.

LOGGING | NOLOGGING specifies the logging attribute. For information about LOGGING and NOLOGGING, see "ALTER TABLE" on page 7-113.

CACHE | NOCACHE For data that will be accessed frequently, specifies whether the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. For information about specifying CACHE or NOCACHE, see "ALTER TABLE" on page 7-113.

LOB_storage_clause specifies the LOB storage characteristics. For information about specifying the parameters of this clause, see "ALTER TABLE" on page 7-113.

modify_LOB_storage_clause modifies the physical attributes of the LOB attribute *lob_item* or LOB object attribute. For information about specifying the parameters of this clause, see "ALTER TABLE" on page 7-113.

partitioning_clauses: The syntax and general functioning of the following partitioning clauses is the same as for the ALTER TABLE statement. See "ALTER TABLE" on page 7-113.

Restrictions:

- You cannot use the *LOB_storage_clause* or *modify_LOB_storage_clause* when modifying a materialized view.
- If you attempt to drop, truncate, or exchange a materialized view partition, Oracle raises an error.

Note: After dropping or truncating a table partition, all materialized views on the table must be refreshed manually. A fast refresh will probably produce incorrect results, but Oracle will not raise an error.

parallel_clause specifies the degree of parallelism for the materialized view. For additional information, see the Notes to the *parallel_clause* of "CREATE TABLE" on page 7-359.

When this clause is set for master tables, performance for materialized view creation and refresh may improve (depending on the materialized view definition query).

NOPARALLEL specifies serial execution. This is the default.

	PARALLEL	causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.
	PARALLEL *integer*	specifies the **degree of parallelism**, which is number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution processes. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

MODIFY PARTITION UNUSABLE LOCAL INDEXES

marks UNUSABLE all the local index partitions associated with *partition*.

MODIFY PARTITION REBUILD UNUSABLE LOCAL INDEXES

rebuilds the unusable local index partitions associated with *partition*.

USING INDEX changes the value of INITRANS, MAXTRANS, and STORAGE parameters for the index Oracle uses to maintain the materialized view's data. If USING INDEX is not specified, then default values are used for the index.

Restriction: You cannot specify the PCTUSED or PCTFREE parameters in this clause.

refresh_clause changes the mode and times for automatic refreshes.

FAST	specifies a fast refresh. A fast refresh uses the materialized view log associated with the detail table or, if you also specify ON DEMAND, with the direct loader log. Oracle creates the direct loader log automatically. No user intervention is needed. Several restrictions exist on the types of materialized views that you can fast refresh. For a complete explanation of when you can fast refresh a materialized view used for replication, see *Oracle8i Replication*. For a complete explanation of when you can fast refresh a materialized view used for data warehousing, see *Oracle8i Tuning*.
COMPLETE	specifies a complete refresh, or a refresh that re-creates the materialized view during each refresh.
FORCE	specifies a fast refresh if one is possible or a complete refresh if a fast refresh is not possible. Oracle decides whether a fast refresh is possible at refresh time.
ON COMMIT	specifies that the refresh is to occur automatically at the next COMMIT operation. **Restriction:** This clause is supported only for materialized views that either include no aggregations or that include no joins. For more information, see *Oracle8i Tuning*.

ON DEMAND — specifies that a refresh will occur when you explicitly invoke a refresh procedure. This method is also called "warehouse refresh", and you can also specify it by calling the DBMS_MVIEW.REFRESH procedure. The types of materialized views you can create by specifying refresh on demand are described in *Oracle8i Tuning*.

Alternatively, this clause specifies that a fast refresh will occur only if you add data using a direct-path method.

If you specify ON COMMIT or ON DEMAND, you cannot also specify START WITH or NEXT.

START WITH — specifies a date expression for the next automatic refresh time.

NEXT — specifies a new date expression for calculating the interval between automatic refreshes.

START WITH and NEXT values must evaluate to times in the future.

WITH PRIMARY KEY — changes a rowid materialized view to a primary key materialized view. Primary key materialized views allow materialized view master tables to be reorganized without affecting the materialized view's ability to continue to fast refresh. The master table must contain an enabled primary key constraint.

For detailed information about primary key materialized views, see *Oracle8i Replication*.

USING ROLLBACK SEGMENT — changes the remote rollback segment to be used during materialized view refresh; *rollback_segment* is the name of the rollback segment to be used.

- DEFAULT specifies that Oracle will choose automatically which rollback segment to use. If you specify DEFAULT, you cannot specify *rollback_segment*.
- MASTER specifies the remote rollback segment to be used at the remote master for the individual materialized view. (To change the local materialized view rollback segment, use the DBMS_REFRESH package, described in *Oracle8i Replication*.)

The master rollback segment is stored on a per-materialized-view basis and is validated during materialized view creation and refresh. If the materialized view is complex, the master rollback segment, if specified, is ignored.

QUERY REWRITE — specifies whether the materialized view is eligible to be used for query rewrite.

ENABLE — enables the materialized view for query rewrite. For more information on query rewrite, see *Oracle8i Concepts*.

Restrictions:

- If the materialized view is in an invalid or unusable state, the ENABLE mode will not take effect until the materialized view is valid and usable.
- You can enable query rewrite only if all user-defined functions in the materialized view are DETERMINISTIC. For more information, see "CREATE FUNCTION" on page 7-266.
- If you use bind variables in a query, the query will not be rewritten to use materialized views even if you enable query rewrite.
- You can enable query rewrite only if the statement contains only repeatable expressions. For example, you cannot include CURRENT_TIME or USER. For more information, see *Oracle8i Tuning*.

DISABLE — specifies that the materialized view is not eligible for use by query rewrite. (If a materialized view is in invalid state, it is not eligible for use by query rewrite, whether or not it is disabled.) However, a disabled materialized view can be refreshed.

COMPILE — explicitly revalidates a materialized view. If an object upon which the materialized view depends is dropped or altered, the materialized view remains accessible, but it is invalidated for purposes of query rewrite. You can use this clause to explicitly revalidate the materialized view to make it eligible for query rewrite.

If the materialized view fails to revalidate, it cannot be either fast refreshed ON DEMAND or used for query rewrite.

Examples

Periodic Refresh Example The following statement changes the automatic refresh mode for the HQ_EMP materialized view to FAST:

```
ALTER SNAPSHOT hq_emp
   REFRESH FAST;
```

The next automatic refresh of the materialized view will be a fast refresh provided it is a simple materialized view and its master table has a materialized view log that was created before the materialized view was created or last refreshed.

Because the REFRESH clause does not specify START WITH or NEXT values, the refresh intervals established by the REFRESH clause when the HQ_EMP materialized view was created or last altered are still used.

NEXT Example The following statement stores a new interval between automatic refreshes for the BRANCH_EMP materialized view:

```
ALTER SNAPSHOT branch_emp
   REFRESH NEXT SYSDATE+7;
```

Because the REFRESH clause does not specify a START WITH value, the next automatic refresh occurs at the time established by the START WITH and NEXT values specified when the BRANCH_EMP materialized view was created or last altered.

At the time of the next automatic refresh, Oracle refreshes the materialized view, evaluates the NEXT expression SYSDATE+7 to determine the next automatic refresh time, and continues to refresh the materialized view automatically once a week.

Because the REFRESH clause does not explicitly specify a refresh mode, Oracle continues to use the refresh mode specified by the REFRESH clause of a previous CREATE MATERIALIZED VIEW or ALTER MATERIALIZED VIEW statement.

Complete Refresh Example The following statement specifies a new refresh mode, next refresh time, and new interval between automatic refreshes of the SF_EMP materialized view:

```
ALTER SNAPSHOT sf_emp
   REFRESH COMPLETE
   START WITH TRUNC(SYSDATE+1) + 9/24
   NEXT SYSDATE+7;
```

The START WITH value establishes the next automatic refresh for the materialized view to be 9:00 a.m. tomorrow. At that point, Oracle performs a complete refresh of the materialized view, evaluates the NEXT expression, and subsequently refreshes the materialized view every week.

Enabling Query Rewrite Example The following statement enables query rewrite on the materialized view MV1 and explicitly revalidates it.

```
ALTER MATERIALIZED VIEW mv1
   ENABLE QUERY REWRITE COMPILE;
```

Rollback Segment Examples The following statement changes the remote master rollback segment used during materialized view refresh to MASTER_SEG:

```
ALTER SNAPSHOT inventory
```

```
   REFRESH USING MASTER ROLLBACK SEGMENT master_seg;
```

The following statement changes the remote master rollback segment used during materialized view refresh to one chosen by Oracle:

```
ALTER SNAPSHOT sales REFRESH USING DEFAULT MASTER ROLLBACK SEGMENT;
```

Primary Key Example The following statement changes a rowid materialized view to a primary key materialized view:

```
ALTER SNAPSHOT emp_rs
   REFRESH WITH PRIMARY KEY;
```

COMPILE Example The following statement recompiles the materialized view STORE_MV:

```
ALTER MATERIALIZED VIEW store_mv COMPILE;
```

Query Rewrite Example The following statement enables query rewrite on the materialized view STORE_MV:

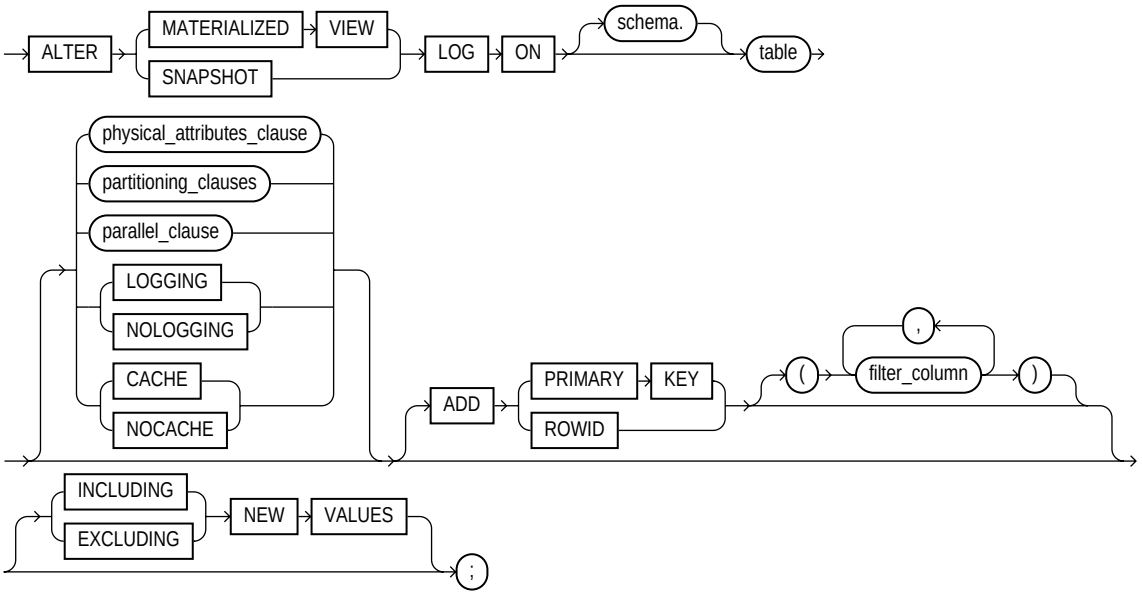
```
ALTER MATERIALIZED VIEW store_mv ENABLE QUERY REWRITE;
```

Modifying Refresh Mode Example The following statement changes the refresh method of materialized view STORE_MV to FAST;

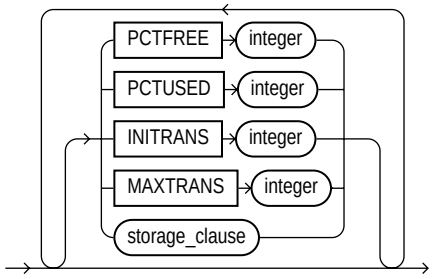
```
ALTER MATERIALIZED VIEW store_mv refresh fast;
```

ALTER MATERIALIZED VIEW LOG / SNAPSHOT LOG

Syntax



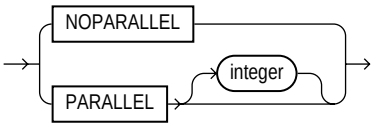
physical_attributes_clause::=



storage_clause: See "storage_clause" on page 7-575.

partitioning_clauses: See "ALTER TABLE" on page 7-113.

parallel_clause::=



Purpose

To alter the storage characteristics, refresh mode or time, or type of an existing materialized view log.

The terms "snapshot" and "materialized view" are synonymous. Both refer to a table that contains the results of a query of one or more tables, each of which may be located on the same or on a remote database. For more information on materialized views, including refreshing them, see "ALTER MATERIALIZED VIEW / SNAPSHOT" on page 7-45. For a description of the various types of materialized views, see "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 7-300.

Prerequisites

Only the owner of the master table or a user with the SELECT privilege for the master table can alter a materialized view log. For detailed information about the prerequisites for ALTER SNAPSHOT LOG, see *Oracle8i Replication*.

Keywords and Parameters

schema	is the schema containing the master table. If you omit *schema*, Oracle assumes the materialized view log is in your own schema.
table	is the name of the master table associated with the materialized view log to be altered.
physical_ attributes_clause	changes the value of PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters for the table, partition, the overflow data segment, or the default characteristics of a partitioned table. For a description of these parameters, see "CREATE TABLE" on page 7-359. See also the "Storage Example" on page 7-57.

partitioning_clauses

The syntax and general functioning of the partitioning clauses is the same as for the ALTER TABLE statement; see "ALTER TABLE" on page 7-113.

Restrictions:

- You cannot use the *LOB_storage_clause* or *modify_LOB_storage_clause* when modifying a materialized view log.
- If you attempt to drop, truncate, or exchange a materialized view log partition, Oracle raises an error.

parallel_clause

specifies the degree of parallelism for the materialized view. For additional information, see the Notes to the *parallel_clause* of "CREATE TABLE" on page 7-359.

When this clause is set for master tables, performance for materialized view creation and refresh may improve (depending on the materialized view definition query).

NOPARALLEL — specifies serial execution. This is the default.

PARALLEL — causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* — specifies the **degree of parallelism**, which is number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution processes. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

LOGGING | NOLOGGING

specifies the logging attribute. For information about specifying this attribute, see "ALTER TABLE" on page 7-113.

CACHE | NOCACHE

for data that will be accessed frequently, specifies whether the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables. For information about specifying CACHE or NOCACHE, see "ALTER TABLE" on page 7-113.

ADD

changes the materialized view log so that it records the primary key values or rowid values when rows in the materialized view master table are updated. This clause can also be used to record additional filter columns.

To stop recording any of this information, you must first drop the materialized view log and then re-create it. Dropping the materialized view log and then re-creating it forces all existing materialized views on the master table to complete refresh.

PRIMARY KEY — specifies that the primary-key values of all rows updated should be recorded in the materialized view log.

ROWID — specifies that the rowid values of all rows updated should be recorded in the materialized view log.

filter_column(s) — are non-primary-key columns referenced by materialized views. For information about filter columns, see *Oracle8i Replication*.

NEW VALUES	specifies whether Oracle saves both old and new values in the materialized view log.	
	INCLUDING	saves old as well as new values in the log. If you are creating a log for a materialized aggregate view with only one master table, and if you want the materialized view to be eligible for fast refresh, you must specify INCLUDING.
	EXCLUDING	saves only new values in the log. This is the default. To save overhead, use this clause for materialized join views and for materialized aggregate views with more than one master table. Such views do not require the old values.

Examples

Storage Example The following statement changes the MAXEXTENTS value of a materialized view log:

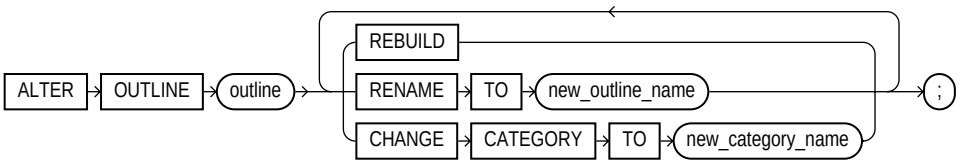
```
ALTER SNAPSHOT LOG ON dept
   STORAGE MAXEXTENTS 50;
```

PRIMARY KEY Example The following statement alters an existing rowid materialized view log to also record primary key information:

```
ALTER SNAPSHOT LOG ON sales
   ADD PRIMARY KEY;
```

ALTER OUTLINE

Syntax



Purpose

To rename a stored outline, reassign it to a different category, or regenerate it by compiling the outline's SQL statement and replacing the old outline data with the outline created under current conditions.

For more information on outlines, see "CREATE OUTLINE" on page 7-323 and *Oracle8i Tuning*.

Prerequisites

To modify an outline, you must have the ALTER ANY OUTLINE system privilege.

Keywords and Parameters

outline	is the name of the outline to be modified.
REBUILD	regenerates the execution plan for *outline* using current conditions.
RENAME TO *new_outline_name*	specifies an outline name to replace *outline*.
CHANGE CATEGORY TO *new_category_name*	specifies the name of the category into which the *outline* will be moved.

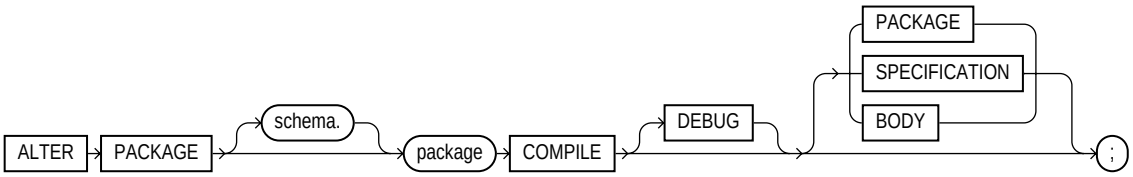
Example

The following statement regenerates a stored outline called SALARIES by compiling the outline's text and replacing the old outline data with the outline created under current conditions.

```
ALTER OUTLINE salaries REBUILD;
```

ALTER PACKAGE

Syntax



Purpose

To explicitly recompile either a package specification, body, or both. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

Because all objects in a package are stored as a unit, the ALTER PACKAGE statement recompiles all package objects together. You cannot use the ALTER PROCEDURE statement or ALTER FUNCTION statement to recompile individually a procedure or function that is part of a package.

> **Note:** This statement does not change the declaration or definition of an existing package. To redeclare or redefine a package, use the "CREATE PACKAGE" or the "CREATE PACKAGE BODY" statement with the OR REPLACE clause.

Prerequisites

The package must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Keywords and Parameters

schema	is the schema containing the package. If you omit *schema*, Oracle assumes the package is in your own schema.
package	is the name of the package to be recompiled.

COMPILE	recompiles the package specification or body. The COMPILE keyword is required. If recompiling the package results in compilation errors, Oracle returns an error and the body remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.
SPECIFICATION	recompiles only the package specification, regardless of whether it is invalid. You might want to recompile a package specification to check for compilation errors after modifying the specification. When you recompile a package specification, Oracle invalidates any local objects that depend on the specification, such as procedures that call procedures or functions in the package. The body of a package also depends on its specification. If you subsequently reference one of these dependent objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.
BODY	recompiles only the package body regardless of whether it is invalid. You might want to recompile a package body after modifying it. Recompiling a package body does not invalidate objects that depend upon the package specification. When you recompile a package body, Oracle first recompiles the objects on which the body depends, if any of those objects are invalid. If Oracle recompiles the body successfully, the body becomes valid.
PACKAGE	recompiles both the package specification and the package body if one exists, regardless of whether they are invalid. This is the default. The recompilation of the package specification and body lead to the invalidation and recompilation as described above for SPECIFICATION and BODY. For information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8i Concepts*.
DEBUG	instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. For information on debugging packages, see *Oracle8i Application Developer's Guide - Fundamentals*.

Examples

This statement explicitly recompiles the specification and body of the ACCOUNTING package in the schema BLAIR:

```
ALTER PACKAGE blair.accounting
   COMPILE PACKAGE;
```

If Oracle encounters no compilation errors while recompiling the ACCOUNTING specification and body, ACCOUNTING becomes valid. BLAIR can subsequently call or reference all package objects declared in the specification of ACCOUNTING without run-time recompilation. If recompiling ACCOUNTING results in compilation errors, Oracle returns an error and ACCOUNTING remains invalid.

Oracle also invalidates all objects that depend upon ACCOUNTING. If you subsequently reference one of these objects without explicitly recompiling it first, Oracle recompiles it implicitly at run time.

To recompile the body of the ACCOUNTING package in the schema BLAIR, issue the following statement:

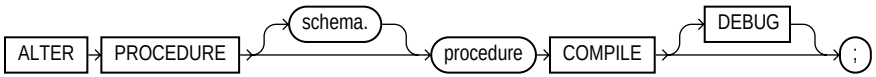
```
ALTER PACKAGE blair.accounting
   COMPILE BODY;
```

If Oracle encounters no compilation errors while recompiling the package body, the body becomes valid. BLAIR can subsequently call or reference all package objects declared in the specification of ACCOUNTING without run-time recompilation. If recompiling the body results in compilation errors, Oracle returns an error message and the body remains invalid.

Because this statement recompiles the body and not the specification of ACCOUNTING, Oracle does not invalidate dependent objects.

ALTER PROCEDURE

Syntax



Purpose

To explicitly recompile a stand-alone stored procedure. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

To recompile a procedure that is part of a package, recompile the entire package using the ALTER PACKAGE statement (see "ALTER PACKAGE" on page 7-59).

> **Note:** This statement does not change the declaration or definition of an existing procedure. To redeclare or redefine a procedure, use the CREATE PROCEDURE statement with the OR REPLACE clause (see "CREATE PROCEDURE" on page 7-333)

The ALTER PROCEDURE statement is quite similar to the ALTER FUNCTION statement (see "ALTER FUNCTION" on page 7-27).

Prerequisites

The procedure must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

Keywords and Parameters

schema	is the schema containing the procedure. If you omit *schema*, Oracle assumes the procedure is in your own schema.
procedure	is the name of the procedure to be recompiled.

COMPILE	causes Oracle to recompile the procedure. The COMPILE keyword is required. Oracle recompiles the procedure regardless of whether it is valid or invalid. ■ Oracle first recompiles objects upon which the procedure depends, if any of those objects are invalid. ■ Oracle also invalidates any local objects that depend upon the procedure, such as procedures that call the recompiled procedure or package bodies that define procedures that call the recompiled procedure. ■ If Oracle recompiles the procedure successfully, the procedure becomes valid. If recompiling the procedure results in compilation errors, then Oracle returns an error and the procedure remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS. For information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8i Concepts*.
DEBUG	instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. For information on debugging procedures, see *Oracle8i Application Developer's Guide - Fundamentals*.

Example

To explicitly recompile the procedure CLOSE_ACCT owned by the user HENRY, issue the following statement:

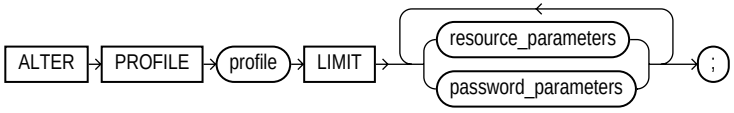
```
ALTER PROCEDURE henry.close_acct
   COMPILE;
```

If Oracle encounters no compilation errors while recompiling CLOSE_ACCT, CLOSE_ACCT becomes valid. Oracle can subsequently execute it without recompiling it at run time. If recompiling CLOSE_ACCT results in compilation errors, Oracle returns an error and CLOSE_ACCT remains invalid.

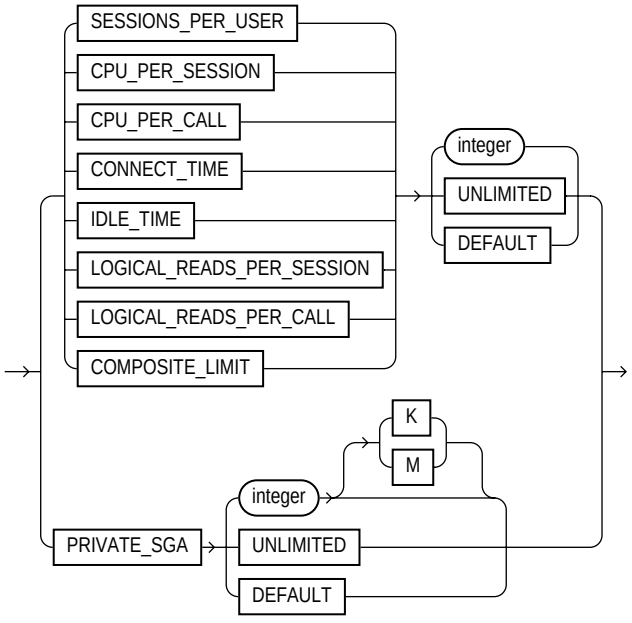
Oracle also invalidates all dependent objects. These objects include any procedures, functions, and package bodies that call CLOSE_ACCT. If you subsequently reference one of these objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.

ALTER PROFILE

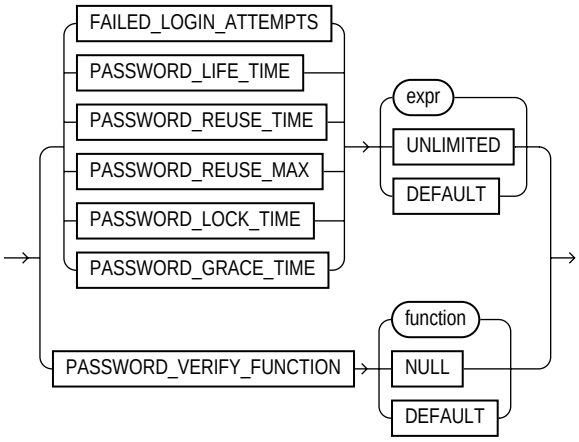
Syntax



resource_parameters::=



password_parameters::=



Purpose

To add, modify, or remove a resource limit or password management parameter in a profile.

Changes made to a profile with an ALTER PROFILE statement affect users only in their subsequent sessions, not in their current sessions.

For information on creating a profile, see "CREATE PROFILE" on page 7-338.

Prerequisites

You must have ALTER PROFILE system privilege to change profile resource limits. To modify password limits and protection, you must have ALTER PROFILE and ALTER USER system privileges.

Keywords and Parameters

The keywords and parameters in the ALTER PROFILE statement all have the same meaning as in the CREATE PROFILE statement. See "CREATE PROFILE" on page 7-338.

Note: You cannot remove a limit from the DEFAULT profile.

Examples

Making a Password Unavailable The following statement makes a password unavailable for reuse for 90 days:

```
ALTER PROFILE prof
   LIMIT PASSWORD_REUSE_TIME 90
   PASSWORD_REUSE_MAX UNLIMITED;
```

Setting Default Values The following statement defaults the PASSWORD_REUSE_TIME value to its defined value in the DEFAULT profile:

```
ALTER PROFILE prof
   LIMIT PASSWORD_REUSE_TIME DEFAULT
   PASSWORD_REUSE_MAX UNLIMITED;
```

Limiting Login Attempts and Password Lock Time The following statement alters profile PROF with FAILED_LOGIN_ATTEMPTS set to 5 and PASSWORD_LOCK_TIME set to 1:

```
ALTER PROFILE prof LIMIT
   FAILED_LOGIN_ATTEMPTS 5
   PASSWORD_LOCK_TIME 1;
```

This statement causes PROF's account to become locked for 1 day after 5 unsuccessful login attempts.

Changing Password Lifetime and Grace Period The following statement modifies profile PROF's PASSWORD_LIFE_TIME to 60 days and PASSWORD_GRACE_TIME to 10 days:

```
ALTER PROFILE prof LIMIT
   PASSWORD_LIFE_TIME 60
   PASSWORD_GRACE_TIME 10;
```

Limiting Concurrent Sessions This statement defines a new limit of 5 concurrent sessions for the ENGINEER profile:

```
ALTER PROFILE engineer LIMIT SESSIONS_PER_USER 5;
```

If the ENGINEER profile does not currently define a limit for SESSIONS_PER_USER, the above statement adds the limit of 5 to the profile. If the profile already defines a limit, the above statement redefines it to 5. Any user assigned the ENGINEER profile is subsequently limited to 5 concurrent sessions.

Removing Limits This statement removes the IDLE_TIME limit from the ENGINEER profile:

```
ALTER PROFILE engineer LIMIT IDLE_TIME DEFAULT;
```

Any user assigned the ENGINEER profile is subject in their subsequent sessions to the IDLE_TIME limit defined in the DEFAULT profile.

Limiting Idle Time This statement defines a limit of 2 minutes of idle time for the DEFAULT profile:

```
ALTER PROFILE default LIMIT IDLE_TIME  2;
```

This IDLE_TIME limit applies to these users:

- Users who are not explicitly assigned any profile
- Users who are explicitly assigned a profile that does not define an IDLE_TIME limit

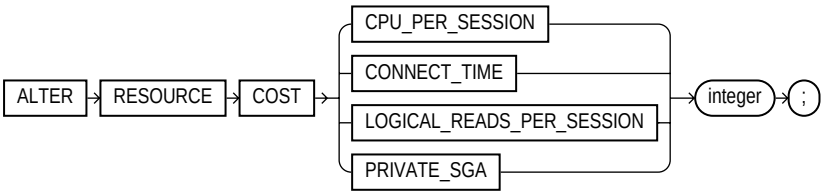
This statement defines unlimited idle time for the ENGINEER profile:

```
ALTER PROFILE engineer LIMIT IDLE_TIME UNLIMITED;
```

Any user assigned the ENGINEER profile is subsequently permitted unlimited idle time.

ALTER RESOURCE COST

Syntax



Purpose

To specify or change the formula by which Oracle calculates the total resource cost used in a session. The weight that you assign to each resource determines how much the use of that resource contributes to the total resource cost. If you do not assign a weight to a resource, the weight defaults to 0 and use of the resource subsequently does not contribute to the cost. The weights you assign apply to all subsequent sessions in the database.

Oracle calculates the total resource cost by first multiplying the amount of each resource used in the session by the resource's weight, and then summing the products for all four resources. For any session, this cost is limited by the value of the COMPOSITE_LIMIT parameter in the user's profile. Both the products and the total cost are expressed in units called **service units**.

Although Oracle monitors the use of other resources, only the four resources shown in the syntax can contribute to the total resource cost for a session. For information on all resources, see "CREATE PROFILE" on page 7-338.

Once you have specified a formula for the total resource cost, you can limit this cost for a session with the COMPOSITE_LIMIT parameter of the CREATE PROFILE statement. If a session's cost exceeds the limit, Oracle aborts the session and returns an error. For information on establishing resource limits, see "CREATE PROFILE" on page 7-338. If you use the ALTER RESOURCE COST statement to change the weight assigned to each resource, Oracle uses these new weights to calculate the total resource cost for all current and subsequent sessions.

Prerequisites

You must have ALTER RESOURCE COST system privilege.

Keywords and Parameters

CPU_PER_SESSION	is the amount of CPU time used by a session measured in hundredth of seconds.
CONNECT_TIME	is the elapsed time of a session measured in minutes.
LOGICAL_READS_PER_ SESSION	is the number of data blocks read during a session, including blocks read from both memory and disk.
PRIVATE_SGA	is the number of bytes of private space in the system global area (SGA) used by a session. This limit applies only if you are using the multi-threaded server architecture and allocating private space in the SGA for your session.
integer	is the weight of each resource.

Example

The following statement assigns weights to the resources CPU_PER_SESSION and CONNECT_TIME:

```
ALTER RESOURCE COST
   CPU_PER_SESSION 100
   CONNECT_TIME      1;
```

The weights establish this cost formula for a session:

```
T = (100 * CPU_PER_SESSION) + (1 * CONNECT_TIME)
```

where the values of CPU_PER_SESSION and CONNECT_TIME are either values in the DEFAULT profile or in the profile of the user of the session.

Because the above statement assigns no weight to the resources LOGICAL_READS_ PER_SESSION and PRIVATE_SGA, these resources do not appear in the formula.

If a user is assigned a profile with a COMPOSITE_LIMIT value of 500, a session exceeds this limit whenever *T* exceeds 500. For example, a session using 0.04 seconds of CPU time and 101 minutes of elapsed time exceeds the limit. A session 0.0301 seconds of CPU time and 200 minutes of elapsed time also exceeds the limit.

You can subsequently change the weights with another ALTER RESOURCE statement:

```
ALTER RESOURCE COST
   LOGICAL_READS_PER_SESSION 2
   CONNECT_TIME 0;
```

These new weights establish a new cost formula:

```
T = (100 * CPU_PER_SESSION) + (2 * LOGICAL_READ_PER_SECOND)
```

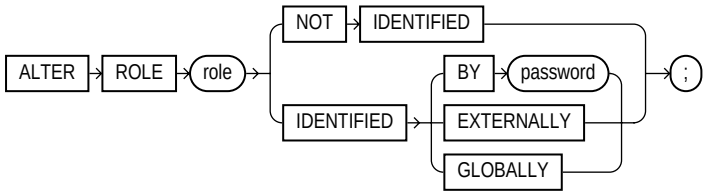
where the values of CPU_PER_SESSION and LOGICAL_READS_PER_SECOND are either the values in the DEFAULT profile or in the profile of the user of this session.

This ALTER RESOURCE COST statement changes the formula in these ways:

- The statement omits a weight for the CPU_PER_SESSION resource and the resource was already assigned a weight, so the resource remains in the formula with its original weight.
- The statement assigns a weight to the LOGICAL_READS_PER_SESSION resource, so this resource now appears in the formula.
- The statement assigns a weight of 0 to the CONNECT_TIME resource, so this resource no longer appears in the formula.
- The statement omits a weight for the PRIVATE_SGA resource and the resource was not already assigned a weight, so the resource still does not appear in the formula.

ALTER ROLE

Syntax



Purpose

To change the authorization needed to enable a role. For information on creating a role, see "CREATE ROLE" on page 7-344. For information on enabling or disabling a role for your session, see "SET ROLE" on page 7-570.

Prerequisites

You must either have been granted the role with the ADMIN OPTION or have ALTER ANY ROLE system privilege.

Before you alter a role to IDENTIFIED GLOBALLY, you must:

- Revoke all grants of roles identified externally to the role and
- Revoke the grant of the role from all users, roles, and PUBLIC.

The one exception to this rule is that you should not revoke the role from the user who is currently altering the role.

Keywords and Parameters

The keywords and parameters in the ALTER ROLE statement all have the same meaning as in the CREATE ROLE statement. See "CREATE ROLE" on page 7-344.

Note: If you have the ALTER ANY ROLE system privilege and you change a role that is IDENTIFIED GLOBALLY to IDENTIFIED BY *password*, IDENTIFIED EXTERNALLY, or NOT IDENTIFIED, then Oracle grants you the altered role with the ADMIN OPTION, as it would have if you had created the role identified nonglobally.

Examples

The following statement changes the role ANALYST to IDENTIFIED GLOBALLY:

```
ALTER ROLE analyst IDENTIFIED GLOBALLY;
```

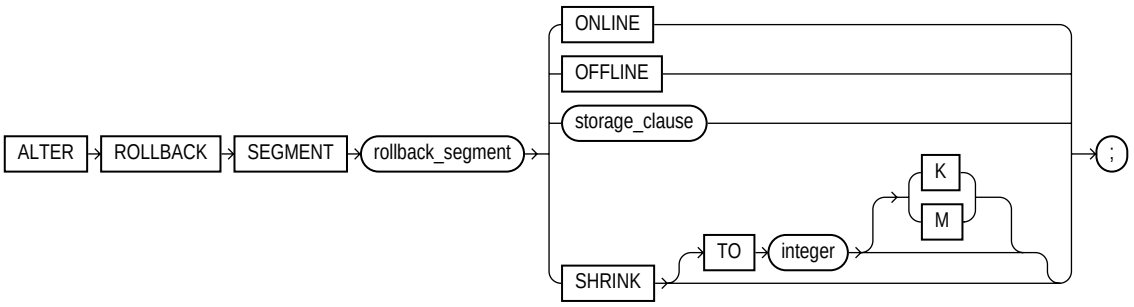
This statement changes the password on the TELLER role to LETTER:

```
ALTER ROLE teller
   IDENTIFIED BY letter;
```

Users granted the TELLER role must subsequently enter the new password "letter" to enable the role.

ALTER ROLLBACK SEGMENT

Syntax



storage_clause: See "storage_clause" on page 7-575.

Purpose

To bring a rollback segment online or offline, to change its storage characteristics, or to shrink it to an optimal or specified size.

For information on creating a rollback segment, see "CREATE ROLLBACK SEGMENT" on page 7-346.

Prerequisites

You must have ALTER ROLLBACK SEGMENT system privilege.

Keywords and Parameters

rollback_segment	specifies the name of an existing rollback segment.
ONLINE	brings the rollback segment online. When you create a rollback segment, it is initially offline and not available for transactions. This clause brings the rollback segment online, making it available for transactions by your instance. You can also bring a rollback segment online when you start your instance with the initialization parameter ROLLBACK_SEGMENTS.

OFFLINE	takes the rollback segment offline. ■ If the rollback segment does not contain any information needed to roll back an active transactions, Oracle takes it offline immediately. ■ If the rollback segment does contain information for active transactions, Oracle makes the rollback segment unavailable for future transactions and takes it offline after all the active transactions are committed or rolled back. Once the rollback segment is offline, it can be brought online by any instance. To see whether a rollback segment is online or offline, query the data dictionary view DBA_ROLLBACK_SEGS. Online rollback segments have a STATUS value of IN_USE. Offline rollback segments have a STATUS value of AVAILABLE. For more information on making rollback segments available and unavailable, see *Oracle8i Administrator's Guide*. **Restriction:** You cannot take the SYSTEM rollback segment offline.
storage_clause	changes the rollback segment's storage characteristics. See the "storage_clause" on page 7-575 for syntax and additional information. **Restriction:** You cannot change the values of the INITIAL and MINEXTENTS for an existing rollback segment.
SHRINK	attempts to shrink the rollback segment to an optimal or specified size. The success and amount of shrinkage depend on the available free space in the rollback segment and how active transactions are holding space in the rollback segment. The value of *integer* is in bytes, unless you specify K or M for kilobytes or megabytes. If you do not specify TO *integer*, then the size defaults to the OPTIMAL value of the *storage_clause* of the CREATE ROLLBACK SEGMENT statement that created the rollback segment. If OPTIMAL was not specified, then the size defaults to the MINEXTENTS value of the *storage_clause* of the CREATE ROLLBACK SEGMENT statement. Regardless of whether you specify TO *integer*: ■ The value to which Oracle shrinks the rollback segment is valid for the execution of the statement. Thereafter, the size reverts to the OPTIMAL value of the CREATE ROLLBACK SEGMENT statement. ■ The rollback segment cannot shrink to less than two extents. To determine the actual size of a rollback segment after attempting to shrink it, query the BYTES, BLOCKS, and EXTENTS columns of the DBA_SEGMENTS view. **Restriction:** For a parallel server, you can shrink only rollback segments that are online to your instance.

Examples

This statement brings the rollback segment RSONE online:

```
ALTER ROLLBACK SEGMENT rsone ONLINE;
```

This statement changes the STORAGE parameters for RSONE:

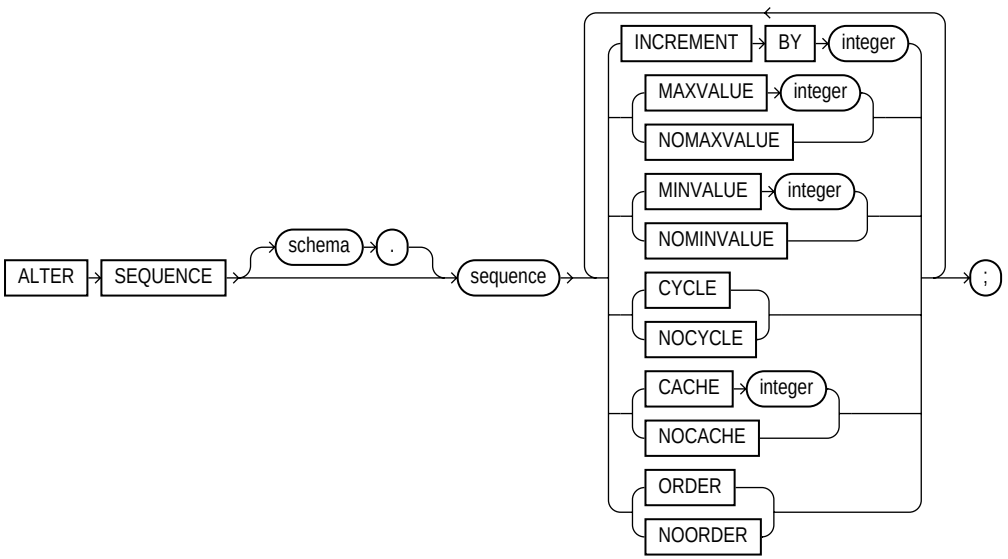
```
ALTER ROLLBACK SEGMENT rsone
   STORAGE (NEXT 1000 MAXEXTENTS 20);
```

This statement attempts to resize a rollback segment to 100 megabytes:

```
ALTER ROLLBACK SEGMENT rsone
   SHRINK TO 100 M;
```

ALTER SEQUENCE

Syntax



Purpose

To change the increment, minimum and maximum values, cached numbers, and behavior of an existing sequence. This statement affects only future sequence numbers. For additional information on sequences, see "CREATE SEQUENCE" on page 7-350.

Prerequisites

The sequence must be in your own schema, or you must have the ALTER object privilege on the sequence, or you must have the ALTER ANY SEQUENCE system privilege.

Keywords and Parameters

The keywords and parameters in this statement serve the same purposes described in "CREATE SEQUENCE" on page 7-350. In addition:

- To restart the sequence at a different number, you must drop and re-create it (see "DROP SEQUENCE" on page 7-471).
- If you change the INCREMENT BY value *before the first invocation of NEXTVAL*, some sequence numbers will be skipped. Therefore, if you want to retain the original START WITH value, you must drop the sequence and re-create it with the original START WITH value and the new INCREMENT BY value.
- Oracle performs some validations. For example, a new MAXVALUE cannot be imposed that is less than the current sequence number.

Examples

This statement sets a new maximum value for the ESEQ sequence:

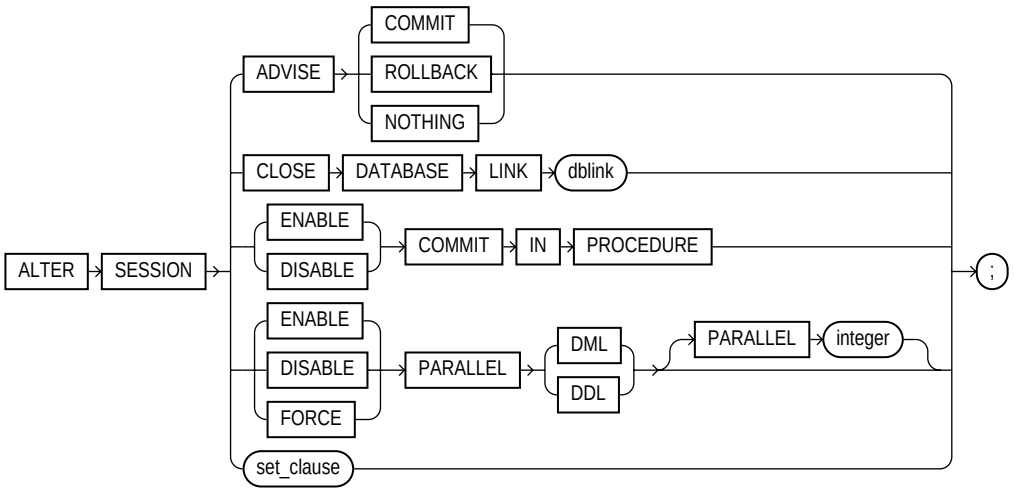
```
ALTER SEQUENCE eseq
   MAXVALUE 1500;
```

This statement turns on CYCLE and CACHE for the ESEQ sequence:

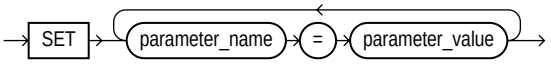
```
ALTER SEQUENCE eseq
   CYCLE
   CACHE 5;
```

ALTER SESSION

Syntax



set_clause::=



Purpose

To specify or modify any of the conditions or parameters that affect your connection to the database. The statement stays in effect until you disconnect from the database.

Prerequisites

To enable and disable the SQL trace facility, you must have ALTER SESSION system privilege.

You do not need any privileges to perform the other operations of this statement unless otherwise indicated.

Keywords and Parameters

ADVISE

sends advice to a remote database to force a distributed transaction. The advice appears in the ADVICE column of the DBA_2PC_PENDING view on the remote database (the value 'C' for COMMIT, 'R' for ROLLBACK, and ' ' for NOTHING). If the transaction becomes in doubt, the administrator of that database can use this advice to decide whether to commit or roll back the transaction.

You can send different advice to different remote databases by issuing multiple ALTER SESSION statements with the ADVISE clause in a single transaction. Each such statement sends advice to the databases referenced in the following statements in the transaction until another such statement is issued. For more information on distributed transactions and how to decide whether to commit or roll back in-doubt distributed transactions, see *Oracle8i Distributed Database Systems*.

CLOSE DATABASE LINK

closes the database link *dblink*. When you issue a statement that uses a database link, Oracle creates a session for you on the remote database using that link. The connection remains open until you end your local session or until the number of database links for your session exceeds the value of the initialization parameter OPEN_LINKS. If you want to reduce the network overhead associated with keeping the link open, use this clause to close the link explicitly if you do not plan to use it again in your session. **You must first close all cursors that use the link and then end your current transaction if it uses the link.**

ENABLE | DISABLE COMMIT IN PROCEDURE

Procedures and stored functions written in PL/SQL can issue COMMIT and ROLLBACK statements. If your application would be disrupted by a COMMIT or ROLLBACK statement not issued directly by the application itself, use the DISABLE form of this clause to prevent procedures and stored functions called during your session from issuing these statements.

You can subsequently allow procedures and stored functions to issue COMMIT and ROLLBACK statements in your session by issuing the ENABLE form of this clause.

Some applications (such as SQL*Forms) automatically prohibit COMMIT and ROLLBACK statements in procedures and stored functions. Refer to your application documentation.

Note: This statement does not apply to database triggers. Triggers can never issue COMMIT or ROLLBACK statements.

PARALLEL DML | DDL

specifies whether all subsequent DML or DDL transactions in the session will be considered for parallel execution. This clause enables you to override the degree of parallelism of tables during the current session without changing the tables themselves. **You can execute this clause only between committed transactions.** Uncommitted transactions must either be committed or rolled back prior to executing this clause.

ENABLE — executes subsequent statements in the session in parallel. This is the default for DDL statements.

- **DML**: executes the session's DML statements in parallel mode if a parallel hint or a parallel clause is specified.
- **DDL**: executes the session's DDL statements in parallel mode if a parallel clause is specified.

Restriction: You cannot specify the optional PARALLEL *integer* with ENABLE.

DISABLE — specifies that subsequent statements will be executed serially. This is the default for DML statements.

- **DML**: executes the session's DML statements serially.
- **DDL**: executes the session's DDL statements serially.

Restriction: You cannot specify the optional PARALLEL *integer* with DISABLE.

FORCE — forces parallel execution of subsequent statements in the session if no parallel DML restrictions are violated (see below). If no parallel clause or hint is specified, then a default degree of parallelism is used. This clause overrides any *parallel_clause* specified in subsequent statements in the session, but is overridden by a parallel hint.

Using FORCE automatically causes all tables created in this session to be created with a default level of parallelism. The effect is the same as if you had specified the *parallel_clause* (with default degree) with the CREATE TABLE statement.

- **DML**: executes subsequent DML statements in the session with the default degree of parallelism, unless a specific degree is specified in this clause.
- **DDL**: executes subsequent DDL statements in the session with the default degree of parallelism, unless a specific degree is specified in this clause. Resulting database objects will have associated with them the prevailing degree of parallelism.
- **PARALLEL** *integer*: explicitly specifies a degree of parallelism, which overrides any *parallel_clause* specified in a subsequent DDL statement in the session, but is overridden by any parallel hint specified in a subsequent DML statement.

The following types of DML operations are not parallelized regardless of this clause:

- operations on clustered tables
- operations with embedded functions that either write or read database or package states
- operations on tables with triggers that could fire
- operations on tables or schema objects containing object types, or LONG or LOB datatypes.

For a detailed description of parallel DML features and hints, see *Oracle8i Tuning*.

set_clause sets the session parameters that follow. You can set values for multiple parameters in the same *set_clause*.

CAUTION: Unless otherwise indicated, the parameters described here are initialization parameters, and the descriptions indicate only the general nature of the parameters. Before changing the values of initialization parameters, please refer to their full description in *Oracle8i Reference* **or** *Oracle8i National Language Support Guide***.**

`CONSTRAINT{S} = {IMMEDIATE | DEFERRED | DEFAULT }`

determines when conditions specified by a deferrable constraint are enforced. `CONSTRAINT{S}` is a session parameter only, not an initialization parameter.

- `IMMEDIATE` indicates that the conditions specified by the deferrable constraint are checked immediately after each DML statement. This setting is equivalent to issuing the SET CONSTRAINTS ALL IMMEDIATE statement at the beginning of each transaction in your session. See the IMMEDIATE parameter of "SET CONSTRAINT(S)" on page 7-568.
- `DEFERRED` indicates that the conditions specified by the deferrable constraint are checked when the transaction is committed. This setting is equivalent to issuing the SET CONSTRAINTS ALL DEFERRED statement at the beginning of each transaction in your session. See the DEFERRED parameter of "SET CONSTRAINT(S)" on page 7-568.
- `DEFAULT` restores all constraints at the beginning of each transaction to their initial state of `DEFERRED` or `IMMEDIATE`.

`CREATE_STORED_OUTLINES = { TRUE | FALSE | 'category_name' }`

determines whether Oracle should automatically create and store an outline for each query submitted during the session. `CREATE_STORED_OUTLINES` is not an initialization parameter.

- `TRUE` enables automatic outline creation for subsequent queries in the same session. These outlines receive a unique system-generated name and are stored in the DEFAULT category. If a particular query already has an outline defined for it in the DEFAULT category, that outline will remain and a new outline will not be created.
- `FALSE` disables automatic outline creation during the session. This is the default.
- *category_name* has the same behavior as TRUE except that any outline created during the session is stored in the *category_name* category.

`CURRENT_SCHEMA = schema`

changes the current schema of the session to the specified schema. Subsequent unqualified references to schema objects during the session will resolve to objects in the specified schema. The setting persists for the duration of the session or until you issue another `ALTER SESSION SET CURRENT_SCHEMA` statement.

This setting offers a convenient way to perform operations on objects in a schema other than that of the current user without having to qualify the objects with the schema name. This setting changes the current schema, but it does not change the session user or the current user, nor does it give you any additional system or object privileges for the session. For more information on this parameter, see *Oracle8i Application Developer's Guide - Fundamentals*.

`DB_BLOCK_CHECKING = {TRUE | FALSE}`

controls whether data block checking is done. The default is `FALSE`, for compatibility with earlier releases where block checking is disabled as a default.

`DB_FILE_MULTIBLOCK_READ_COUNT = integer`

specifies with *integer* the maximum number of blocks read in one I/O operation during a sequential scan. The default is 8.

`FAST_START_IO_TARGET`

specifies the target number of IOs (reads and writes) to and from buffer cache that Oracle should perform upon crash or instance recovery. Oracle continuously calculates the actual number of IOs that would be needed for recovery and compares that number against the target. If the actual number is greater than the target, Oracle attempts to write additional dirty buffers to advance the checkpoint, while minimizing the affect on performance.

For information on how to tune this parameter, see *Oracle8i Tuning*.

`FLAGGER = { ENTRY | INTERMEDIATE | FULL | OFF }`

specifies FIPS flagging, which causes an error message to be generated when a SQL statement issued is an extension of ANSI SQL92. In Oracle, there is currently no difference between Entry, Intermediate, or Full level flagging. Once flagging is set in a session, a subsequent ALTER SESSION SET FLAGGER statement will work, but generates the message, ORA-00097. This allows FIPS flagging to be altered without disconnecting the session. OFF turns off flagging.

`GLOBAL_NAMES = { TRUE | FALSE }`

When you start an instance, Oracle determines whether to enforce global name resolution for remote objects accessed in SQL statements based on the value of the initialization parameter GLOBAL_NAMES. This parameter enables or disables global name resolution for the duration of the session. TRUE enables the enforcement of global names. FALSE disables the enforcement of global names. You can also enable or disable global name resolution for your instance with the GLOBAL_NAMES parameter of the ALTER SYSTEM statement.

Oracle recommends that you enable global name resolution if you use or plan to use distributed processing. For more information on global name resolution and how Oracle enforces it, see "Referring to Objects in Remote Databases" on page 2-74 and *Oracle8i Distributed Database Systems*.

`HASH_AREA_SIZE = integer`

specifies in bytes the amount of memory to use for hash join operations. The default is twice the value of the SORT_AREA_SIZE initialization parameter.

HASH_JOIN_ENABLED = {TRUE | FALSE}

enables or disables the use of the hash join operation in queries. The default is TRUE, which enables hash joins.

HASH_MULTIBLOCK_IO_COUNT = integer

specifies the number of data blocks to read and write during a hash join operation. The value multiplied by the DB_BLOCK_SIZE initialization parameter should not exceed 64 K. The default value for this parameter is 1. If the multi-threaded server is used, the value is always 1, and any value specified here is ignored.

INSTANCE = integer

in a parallel server, accesses database files as if the session were connected to the instance specified by *integer*. INSTANCE is a session parameter only, not an initialization parameter. For optimum performance, each instance of a parallel server uses its own private rollback segments, freelist groups, and so on. In a parallel server, you normally connect to a particular instance and access data that is partitioned primarily for your use. If you must connect to another instance, the data partitioning can be lost. Setting this parameter lets you access an instance as if you were connected to your own instance.

ISOLATION_LEVEL = { SERIALIZABLE | READ COMMITTED }

specifies how transactions containing database modifications are handled. ISOLATION_LEVEL is a session parameter only, not an initialization parameter.

- SERIALIZABLE indicates that transactions in the session use the serializable transaction isolation mode as specified in SQL92. That is, if a serializable transaction attempts to execute a DML statement that updates rows currently being updated by another uncommitted transaction at the start of the serializable transaction, then the DML statement fails. A serializable transaction can see its own updates.
- READ COMMITTED indicates that transactions in the session will use the default Oracle transaction behavior. Thus, if the transaction contains DML that requires row locks held by another transaction, then the DML statement will wait until the row locks are released.

LOG_ARCHIVE_DEST_n = {null_string | {LOCATION=pathname | SERVICE=servicename}
[MANDATORY | OPTIONAL] [REOPEN[=retry_time_in_seconds]]}

specifies up to five session-specific valid operating system pathnames or Oracle service names (plus other related options) as destinations for archive redo log file groups (*n* = integers 1 through 5). For a description of the options, refer to *Oracle8i Reference*.

Restrictions: If you set a value for this parameter,

- You cannot have definitions for the parameters LOG_ARCHIVE_DEST and LOG_ARCHIVE_DUPLEX_DEST in your initialization parameter file, nor can you set values for those parameters with the ALTER SYSTEM statement.
- You cannot start archiving to a specific location using the ALTER SYSTEM ARCHIVE LOG TO *location* statement.

`LOG_ARCHIVE_DEST_STATE_n = {ENABLE | DEFER}`

specifies the session-specific state associated with the corresponding `LOG_ARCHIVE_DEST_n` parameter.

- `ENABLE` specifies that any associated valid destination can be used for archiving. This is the default.
- `DEFER` specifies that Oracle will not consider for archiving any destination associated with the corresponding `LOG_ARCHIVE_DEST_n` parameter.

`LOG_ARCHIVE_MIN_SUCCEED_DEST = integer`

specifies the session-specific minimum number of destinations that must succeed in order for the online log file to be available for reuse.

`MAX_DUMP_FILE_SIZE = { size | UNLIMITED }`

specifies the upper limit of trace dump file size. Specify the maximum *size* as either a nonnegative integer that represents the number of blocks, or as `UNLIMITED`. If you specify `UNLIMITED`, no upper limit is imposed.

NLS parameters: When you start an instance, Oracle establishes support based on the values of initialization parameters that begin with "NLS". You can query the dynamic performance table V$NLS_PARAMETERS to see the current NLS attributes for your session. For more information about NLS parameters, see *Oracle8i National Language Support Guide*.

`NLS_CALENDAR = 'text'`

explicitly specifies a new calendar type.

`NLS_COMP = 'text'`

specifies that linguistic comparison is to be used according to the NLS_SORT parameter. This parameter obviates the need to specify NLS_SORT in SQL statements.

`NLS_CURRENCY = 'text'`

explicitly specifies a new value for the L number format element (the local currency symbol). The symbol cannot exceed 10 characters.

`NLS_DATE_FORMAT = 'fmt'`

explicitly specifies a new default date format. The '*fmt*' value must be a date format model as specified in the section "Date Format Elements" on page 2-40.

`NLS_DATE_LANGUAGE = language`

explicitly changes the language for names and abbreviations of days and months, and for spelled-out values of other date format elements.

`NLS_ISO_CURRENCY = territory`

explicitly specifies the territory whose ISO currency symbol should be used. That territory's currency symbol then becomes the value of the C number format element.

`NLS_LANGUAGE = language`

changes the language in which Oracle returns errors and other messages. This parameter also implicitly specifies new values for these items:

- language for day and month names and abbreviations and spelled values of other elements
- linguistic sort sequences or binary sorts
- B.C. and A.D. indicators
- A.M. and P.M. meridian indicators

`NLS_NUMERIC_CHARACTERS = 'text'`

explicitly specifies a new decimal character and group separator. The '*text*' value must have this form:

```
'dg'
```

where: *d* is the new decimal character, and *g* is the new group separator.

The decimal character and the group separator must be two different single-byte characters, and cannot be a numeric value or any of the following characters: plus sign ("+"), minus sign or hyphen ("–"), less-than sign ("<"), or greater-than sign (">").

If the decimal character is not a period (.), you must use single quotation marks to enclose all number values that appear in expressions in your SQL statements. When not using a period for the decimal point, use the TO_NUMBER function to ensure that a valid number is retrieved.

`NLS_SORT = { sort | BINARY}`

changes the sequence into which Oracle sorts character values. *sort* specifies the name of a linguistic sort sequence. BINARY specifies a binary sort. The default is BINARY.

`NLS_TERRITORY = territory`

implicitly specifies new values for these items:

- default date format
- decimal character and group separators
- local currency symbol
- ISO currency symbol
- first day of the week for D date format element

`NLS_DUAL_CURRENCY = 'text'`

explicitly specifies a new "Euro" (or other) dual currency symbol. The value of *text* is returned by the number format element U (see "Number Format Elements" on page 2-36); *text* cannot exceed 10 characters.

`OBJECT_CACHE_MAX_SIZE_PERCENT = integer`

specifies the percentage of the optimal cache size that the session object cache can grow beyond the optimal size. The default is 10.

`OBJECT_CACHE_OPTIMAL_SIZE = integer`

specifies (in kilobytes) the size to which the session object cache is reduced when it exceeds maximum size. The default is 100.

`OPTIMIZER_INDEX_CACHING = integer`

lets you tune the optimizer to favor nested loops joins. The value of *integer* indicates the percentage of the index blocks assumed to be in the cache.

`OPTIMIZER_INDEX_COST_ADJ = integer`

let you tune optimizer behavior for access path selection to make the optimizer more likely to select an index access path than a full table scan. The value of *integer* is a percentage indicating the importance the optimizer attaches to the index path compared with "normal". The default is 100 (indicating 100%), which makes the optimizer cost index access paths at the regular cost.

`OPTIMIZER_MAX_PERMUTATIONS = integer`

lets you limit the amount of work the optimizer expends on optimizing queries with large joins. The value of *integer* is the number of permutations of the tables the optimizer will consider with large joins.

`OPTIMIZER_MODE = { ALL_ROWS | FIRST_ROWS | RULE | CHOOSE }`

specifies the approach and mode of the optimizer for your session. For information on how to choose a goal for the cost-based approach based on the characteristics of your application, see *Oracle8i Concepts* and *Oracle8i Tuning*.

- `ALL_ROWS` specifies the cost-based approach and optimizes for best throughput.
- `FIRST_ROWS` specifies the cost-based approach and optimizes for best response time.
- `RULE` specifies the rule-based approach. (The rule-based optimizer does not use function-based indexes.)
- `CHOOSE` causes the optimizer to choose an optimization approach based on the presence of statistics in the data dictionary.

`OPTIMIZER_PERCENT_PARALLEL = integer`

specifies the amount of parallelism the optimizer uses in its cost functions. The default is 0 (no parallelism).

`OPTIMIZER_SEARCH_LIMIT = integer`

specifies the search limit for the optimizer. The default is 5.

`PARALLEL_BROADCAST_ENABLED = { TRUE | FALSE }`

lets you enhance performance during hash and merge joins.

`PARALLEL_INSTANCE_GROUP = ' text '`

identifies the parallel instance group to be used for spawning parallel query slaves. The default is all active instances. **Set this parameter only if you are running Oracle Parallel Server in parallel mode.**

`PARALLEL_MIN_PERCENT = integer`

specifies the minimum percent of threads required for parallel query. The default is 0 (no parallelism).

`PARTITION_VIEW_ENABLED = { TRUE | FALSE }`

When set to TRUE, this parameter causes the optimizer to skip unnecessary table accesses in a partition view.

Note: For important information on partition views, see "Partition Views" on page 7-431.

`PLSQL_V2_COMPATIBILITY = { TRUE | FALSE }`

if TRUE, modifies the compile-time behavior of PL/SQL programs to allow language constructs that are illegal in Oracle8 and Oracle8*i* (PL/SQL V3), but were legal in Oracle7 (PL/SQL V2). FALSE disallows illegal Oracle7 PL/SQL V2 constructs. This is the default.

See the *PL/SQL User's Guide and Reference* and *Oracle8i Reference* for more information about this session parameter.

`REMOTE_DEPENDENCIES_MODE = { TIMESTAMP | SIGNATURE }`

specifies how dependencies of remote stored procedures are handled by the session. For more information, refer to *Oracle8i Application Developer's Guide - Fundamentals.*

`QUERY_REWRITE_ENABLED = { TRUE | FALSE }`

enables or disables query rewrite on all materialized views that have not been explicitly disabled. Query rewrite is disabled by default. It is also disabled by rule-based optimization (that is, if the `OPTIMIZER_MODE` parameter is set to `RULE`).

This parameter has the following additional effect on the use of function-based indexes:

- If this parameter is set to `TRUE`, Oracle will use function-based indexes to derive values of SQL expressions. If in addition the `QUERY_REWRITE_INTEGRITY` parameter is set to any value other than `ENFORCED`, Oracle will derive such values even if the index is based on a user-defined (rather than SQL) function.
- If this parameter is set to `FALSE`, Oracle will not use function-based indexes to derive values of SQL expressions, but it will use such indexes to obtain values of real columns in the index.

Enabling or disabling query rewrite does not affect descending indexes.

For more information on query rewrite, see *Oracle8i Tuning*.

`QUERY_REWRITE_INTEGRITY = { ENFORCED | TRUSTED | STALE_TOLERATED }`

sets the minimum consistency level for query rewrite. The following values are permitted:

- `ENFORCED` is the safest level. It relies only on system-enforced relationships so that data integrity and correctness can be guaranteed. This level ensures that query rewrite will not use any function-based index or any materialized view that includes a call to a user-defined function.

  In addition, this level ensures that query rewrite will not use any dimensional information or any constraints enabled with the RELY keyword.
- `TRUSTED` specifies that materialized views created with the ON PREBUILT TABLE clause are supported, and trusted but unenforced join relationships are accepted. Query rewrite uses join information from dimensions and enables unenforced constraints with the RELY keyword.
- `STALE_TOLERATED` specifies that any stale, usable materialized view may be used.

This parameter does not affect descending indexes.

For more information on query rewrite integrity level, see *Oracle8i Tuning*. For information on dimensions, see "CREATE DIMENSION" on page 7-259. For information on constraints enabled with the RELY keyword, see "constraint_clause" on page 7-217.

`SESSION_CACHED_CURSORS = integer`

specifies the number of frequently used cursors that can be retained in the cache. The cursors can be open or closed, which is particularly useful for Oracle tools that close all session cursors associated with a form when switching to another form. In such cases, frequently used cursors do not have to be reparsed. A least recently used algorithm ages out entries in the cache to make room for new entries when needed. For more information on session cursor caching, see *Oracle8i Tuning*.

`SKIP_UNUSABLE_INDEXES = { TRUE | FALSE }`

controls the use and reporting of tables with unusable indexes or index partitions.

- `TRUE` disables error reporting of indexes marked UNUSABLE. Allows inserts, deletes, and updates to tables with unusable indexes or index partitions.
- `FALSE` enables error reporting of indexes marked UNUSABLE. Does not allow inserts, deletes, and updates to tables with unusable indexes or index partitions. This is the default.

`SORT_AREA_RETAINED_SIZE = integer`

specifies (in bytes) the maximum amount of memory that each sort operation will retain after the first fetch is done, until the cursor ends. The default is the value of the SORT_AREA_SIZE parameter.

`SORT_AREA_SIZE = integer`

specifies (in bytes) the maximum amount of memory to use for each sort operation. The default is OS-dependent.

`SORT_MULTIBLOCK_READ_COUNT = integer`

specifies the number of database blocks to read each time a sort performs a read from temporary segments. The default is 2.

`SQL_TRACE = { TRUE | FALSE }`

The SQL trace facility generates performance statistics for the processing of SQL statements. When you begin a session, Oracle enables or disables the SQL trace facility based on the value of this parameter. You can subsequently enable or disable the SQL trace facility for your own session with the SQL_TRACE parameter of the ALTER SESSION statement. TRUE enables the SQL trace facility. FALSE disables it.

For more information on the SQL trace facility, including how to format and interpret its output, see *Oracle8i Tuning*.

`STAR_TRANSFORMATION_ENABLED = { TRUE | FALSE }`

determines whether a cost-based query transformation will be applied to star queries. The default is FALSE.

`TIMED_STATISTICS = {TRUE | FALSE }`

specifies whether the server requests the time from the operating system when generating time-related statistics. The default is FALSE.

`USE_STORED_OUTLINES = { TRUE | FALSE | 'category_name' }`

determines whether the optimizer will use stored outlines to generate execution plans. `USE_STORED_OUTLINES` is not an initialization parameter.

- `TRUE` causes the optimizer to use outlines stored in the DEFAULT category when compiling requests.
- `FALSE` specifies that the optimizer should not use stored outlines. This is the default.
- *category_name* causes the optimizer to use outlines stored in the *category_name* category when compiling requests.

Examples

PARALLEL Example Issue the following statement to enable parallel DML mode for the current session:

```
ALTER SESSION ENABLE PARALLEL DML;
```

ADVISE Example The following transaction inserts an employee record into the EMP table on the database identified by the database link SITE1 and deletes an employee record from the EMP table on the database identified by SITE2:

```
ALTER SESSION
   ADVISE COMMIT;

INSERT INTO emp@site1
   VALUES (8002, 'FERNANDEZ', 'ANALYST', 7566,
   TO_DATE('04-OCT-1992', 'DD-MON-YYYY'), 3000, NULL, 20);

ALTER SESSION
   ADVISE ROLLBACK;

DELETE FROM emp@site2
   WHERE empno = 8002;

COMMIT;
```

This transaction has two ALTER SESSION statements with the ADVISE clause. If the transaction becomes in doubt, SITE1 is sent the advice 'COMMIT' by virtue of the first ALTER SESSION statement and SITE2 is sent the advice 'ROLLBACK' by virtue of the second.

CLOSE DATABASE LINK Example This statement updates the employee table on the SALES database using a database link, commits the transaction, and explicitly closes the database link:

```
UPDATE emp@sales
   SET sal = sal + 200
   WHERE empno = 9001;

COMMIT;

ALTER SESSION
   CLOSE DATABASE LINK sales;
```

Date Format Example The following statement dynamically changes the default date format for your session to 'YYYY MM DD-HH24:MI:SS':

```
ALTER SESSION
   SET NLS_DATE_FORMAT = 'YYYY MM DD HH24:MI:SS';
```

Oracle uses the new default date format:

```
SELECT TO_CHAR(SYSDATE) Today
```

```
   FROM DUAL;

TODAY
-------------------
1997 08 12 14:25:56
```

Date Language Example The following statement changes the language for date format elements to French:

```
ALTER SESSION
   SET NLS_DATE_LANGUAGE = French;

SELECT TO_CHAR(SYSDATE, 'Day DD Month YYYY') Today
   FROM DUAL;

TODAY
---------------------------
Mardi    28 Février   1997
```

ISO Currency Example The following statement dynamically changes the ISO currency symbol to the ISO currency symbol for the territory America:

```
ALTER SESSION
   SET NLS_ISO_CURRENCY = America;

SELECT TO_CHAR( SUM(sal), 'C999G999D99') Total
   FROM emp;

TOTAL
-------------
USD29,025.00
```

Decimal Character and Group Separator Example The following statement dynamically changes the decimal character to comma (,) and the group separator to period (.):

```
ALTER SESSION SET NLS_NUMERIC_CHARACTERS = ',.' ;
```

Oracle returns these new characters when you use their number format elements:

```
SELECT TO_CHAR( SUM(sal), 'L999G999D99') Total FROM emp ;

TOTAL
-------------
FF29.025,00
```

NLS Currency Example The following statement dynamically changes the local currency symbol to 'DM':

```
ALTER SESSION
   SET NLS_CURRENCY = 'DM';

SELECT TO_CHAR( SUM(sal), 'L999G999D99') Total
   FROM emp;

TOTAL
-------------
DM29.025,00
```

NLS Language Example The following statement dynamically changes to French the language in which error messages are displayed:

```
ALTER SESSION
   SET NLS_LANGUAGE = FRENCH;

SELECT * FROM DMP;

ORA-00942: Table ou vue inexistante
```

Linguistic Sort Example The following statement dynamically changes the linguistic sort sequence to Spanish:

```
ALTER SESSION
   SET NLS_SORT = XSpanish;
```

Oracle sorts character values based on their position in the Spanish linguistic sort sequence.

SQL Trace Example To enable the SQL trace facility for your session, issue the following statement:

```
ALTER SESSION
   SET SQL_TRACE = TRUE;
```

Query Rewrite Example This statement enables query rewrite in the current session for all materialized views that have not been explicitly disabled:

```
ALTER SESSION SET QUERY_REWRITE_ENABLED = TRUE;
```

ALTER SNAPSHOT

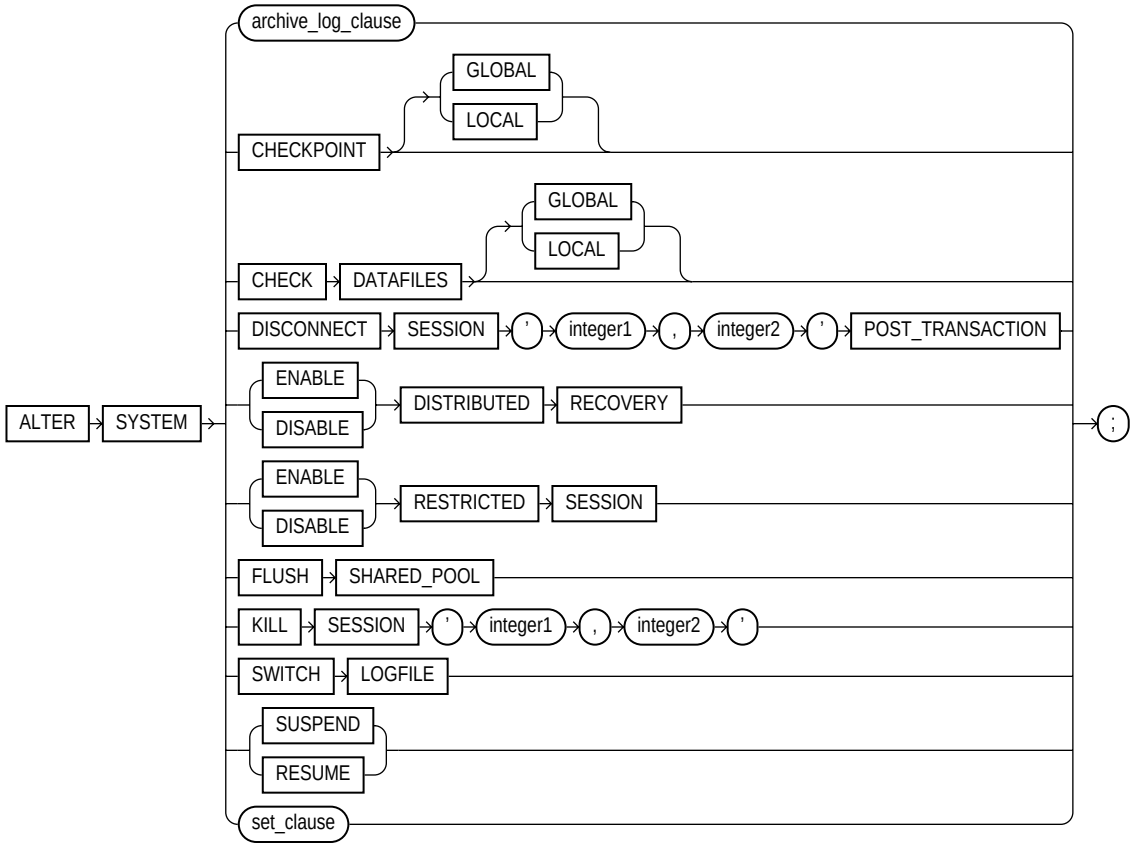
In Oracle8*i*, "snapshots" are synonymous with "materialized views." Please see "ALTER MATERIALIZED VIEW / SNAPSHOT" on page 7-45.

ALTER SNAPSHOT LOG

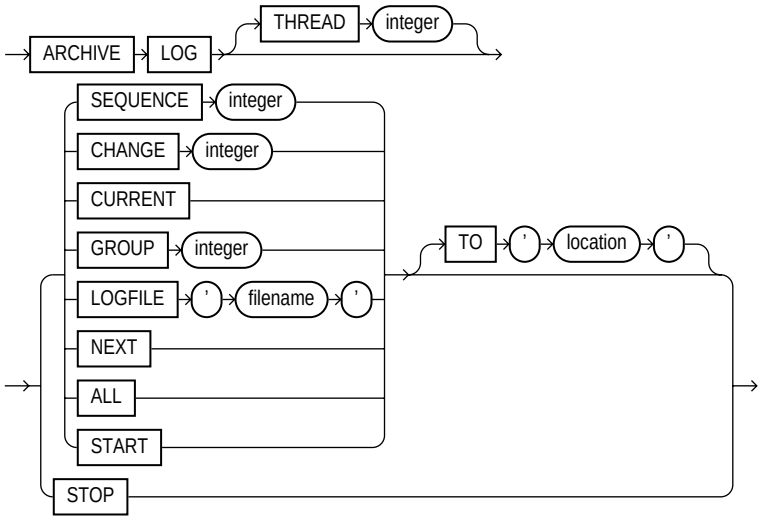
In Oracle8*i*, "snapshots" are synonymous with "materialized views." Please see "ALTER MATERIALIZED VIEW LOG / SNAPSHOT LOG" on page 7-54.

ALTER SYSTEM

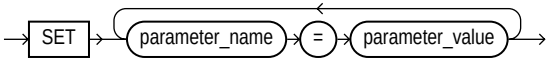
Syntax



archive_log_clause::=



set_clause::=



Purpose

To dynamically alter your Oracle instance. The settings stay in effect as long as the database is mounted.

Prerequisites

You must have ALTER SYSTEM system privilege.

To specify the *archive_log_clause*, you must have the OSDBA or OSOPER role enabled.

Keywords and Parameters

archive_log_clause	manually archives redo log files or enables or disables automatic archiving. To use this clause, your instance must have the database mounted. The database can be either open or closed unless otherwise noted.

Notes:

- You can also manually archive redo log file groups with the ARCHIVE LOG SQL*Plus statement. For information on this statement, see the *SQL*Plus User's Guide and Reference*.
- You can also have Oracle archive redo log files groups automatically. For information on automatic archiving, see *Oracle8i Administrator's Guide*. You can always manually archive redo log file groups regardless of whether automatic archiving is enabled.

THREAD	specifies the thread containing the redo log file group to be archived. **Set this parameter only if you are using Oracle with the Parallel Server option in parallel mode.**
SEQUENCE	manually archives the online redo log file group identified by the log sequence number *integer* in the specified thread. If you omit the THREAD parameter, Oracle archives the specified group from the thread assigned to your instance.
CHANGE	manually archives the online redo log file group containing the redo log entry with the system change number (SCN) specified by *integer* in the specified thread. If the SCN is in the current redo log file group, Oracle performs a log switch. If you omit the THREAD parameter, Oracle archives the groups containing this SCN from all enabled threads. **You can use this clause only when your instance has the database open.**
CURRENT	manually archives the current redo log file group of the specified thread, forcing a log switch. If you omit the THREAD parameter, Oracle archives all redo log file groups from all enabled threads, including logs previous to current logs. **You can use this clause only when your instance has the database open.**

Note: If you specify a redo log file group for archiving with the CHANGE or CURRENT clause, and earlier redo log file groups are not yet archived, Oracle archives all unarchived groups up to and including the specified group.

GROUP	manually archives the online redo log file group with the GROUP value specified by *integer*. You can determine the GROUP value for a redo log file group by examining the data dictionary view DBA_LOG_FILES. If you specify both the THREAD and GROUP parameters, the specified redo log file group must be in the specified thread.
LOGFILE	manually archives the online redo log file group containing the redo log file member identified by '*filename*'. If you specify both the THREAD and LOGFILE parameters, the specified redo log file group must be in the specified thread. **Restriction:** You must archive redo log file groups in the order in which they are filled. If you specify a redo log file group for archiving with the LOGFILE parameter, and earlier redo log file groups are not yet archived, Oracle returns an error.

	NEXT	manually archives the next online redo log file group from the specified thread that is full but has not yet been archived. If you omit the THREAD parameter, Oracle archives the earliest unarchived redo log file group from any enabled thread.
	ALL	manually archives all online redo log file groups from the specified thread that are full but have not been archived. If you omit the THREAD parameter, Oracle archives all full unarchived redo log file groups from all enabled threads.
	START	enables automatic archiving of redo log file groups. **Restriction:** You can enable automatic archiving only for the thread assigned to your instance.
	TO '*location*'	specifies the primary location to which the redo log file groups are archived. The value of this parameter must be a fully specified file location following the conventions of your operating system. If you omit this parameter, Oracle archives the redo log file group to the location specified by the initialization parameters `LOG_ARCHIVE_DEST` or `LOG_ARCHIVE_DEST_`*n*. **Note:** You can enhance recovery reliability by setting the related archive parameters `LOG_ARCHIVE_DEST_DUPLEX` and `LOG_ARCHIVE_MIN_SUCCEED_DEST`.
	STOP	disables automatic archiving of redo log file groups. You can disable automatic archiving only for the thread assigned to your instance.
CHECKPOINT	explicitly forces Oracle to perform a checkpoint, ensuring that all changes made by committed transactions are written to datafiles on disk. You can specify this clause only when your instance has the database open. Oracle does not return control to you until the checkpoint is complete.	
	GLOBAL	in an Oracle Parallel Server environment, performs a checkpoint for all instances that have opened the database. This is the default.
	LOCAL	in an Oracle Parallel Server environment, performs a checkpoint only for the thread of redo log file groups for your instance.
	For more information on checkpoints, see *Oracle8i Concepts*.	
CHECK DATAFILES	in a distributed database system, such as an Oracle Parallel Server environment, updates an instance's SGA from the database control file to reflect information on all online datafiles.	
	GLOBAL	performs this synchronization for all instances that have opened the database. This is the default.
	LOCAL	performs this synchronization only for the local instance.
	Your instance should have the database open. For more information, see *Oracle8i Parallel Server Concepts and Administration*.	

DISCONNECT SESSION ... POST_ TRANSACTION

disconnects the current session by destroying the dedicated server process (or virtual circuit if the connection was made by way of a multi-threaded server). This clause allows ongoing transactions to complete before the session is disconnected, in contrast to the KILL SESSION clause.To use this clause, your instance must have the database open.

If system parameters are appropriately configured, application failover will take effect. For more information about application failover see *Oracle8i Tuning* and *Oracle8i Parallel Server Concepts and Administration*. You must identify the session with both of the following values from the V$SESSION view:

integer1 is the value of the SID column.

integer2 is the value of the SERIAL# column.

DISTRIBUTED RECOVERY

specifies whether or not distributed recovery is enabled. To use this clause, your instance must have the database open.

ENABLE enables distributed recovery. In a single-process environment, you must use this clause to initiate distributed recovery.

You may need to issue the ENABLE DISTRIBUTED RECOVERY statement more than once to recover an in-doubt transaction if the remote node involved in the transaction is not accessible. In-doubt transactions appear in the data dictionary view DBA_2PC_PENDING. For more information about distributed transactions and distributed recovery, see *Oracle8i Distributed Database Systems*.

DISABLE disables distributed recovery.

RESTRICTED SESSION

specifies whether logon to Oracle is restricted

ENABLE allows only users with RESTRICTED SESSION system privilege to log on to Oracle. Existing sessions are not terminated.

DISABLE reverses the effect of the ENABLE RESTRICTED SESSION clause, allowing all users with CREATE SESSION system privilege to log on to Oracle. This is the default.

You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

FLUSH SHARED_POOL

clears all data from the shared pool in the system global area (SGA). The shared pool stores

- cached data dictionary information and
- shared SQL and PL/SQL areas for SQL statements, stored procedures, function, packages, and triggers.

This statement does not clear shared SQL and PL/SQL areas for items that are currently being executed. You can use this clause regardless of whether your instance has the database dismounted or mounted, open or closed.

KILL SESSION terminates a session, rolls back ongoing transactions, releases all session locks, and frees all session resources. To use this clause, your instance must have the database open. You must identify the session with both of the following values from the V$SESSION view:

integer1 is the value of the SID column.

integer2 is the value of the SERIAL# column.

If the session is performing some activity that must be completed, such as waiting for a reply from a remote database or rolling back a transaction, Oracle waits for this activity to complete, kills the session, and then returns control to you. If the waiting lasts a minute, Oracle marks the session to be killed and returns control to you with a message that the session is marked to be killed. Oracle then kills the session when the activity is complete.

Restriction: You can kill a session only on the same instance as your current session.

SWITCH LOGFILE explicitly forces Oracle to begin writing to a new redo log file group, regardless of whether the files in the current redo log file group are full. When you force a log switch, Oracle begins to perform a checkpoint. Oracle returns control to you immediately rather than when the checkpoint is complete. To use this clause, your instance must have the database open.

SUSPEND suspends all I/O (datafile, control file, and file header) as well as queries, in all instances, enabling you to make copies of the database without having to handle ongoing transactions.

Restrictions:

- Do not use this clause unless you have put the database tablespaces in hot backup mode.
- If you start a new instance while the system is suspended, that new instance will not be suspended.

RESUME makes the database available once again for queries and I/O.

For more information on the SUSPEND clause and RESUME clause, refer to *Oracle8i Backup and Recovery Guide*.

set_clause sets the system parameters that follow. You can set values for multiple parameters in the same *set_clause*.

The DEFERRED keyword sets or modifies the value of the parameter for future sessions that connect to the database.

CAUTION: Unless otherwise noted, these parameters are initialization parameters, and the descriptions provided here indicate only the general nature of the parameters. Before changing the values of initialization parameters, please refer to their full description in *Oracle8i Reference* **and** *Oracle8i National Language Support Guide***.**

```
AQ_TM_PROCESSES = integer
```

is an Advanced Queuing parameter that specifies whether a time manager process is created. Accepted values are 1 (creates one time manager process to monitor messages) and 0 (does not create a time manager process).

BACKGROUND_DUMP_DEST = 'text'

specifies the pathname for a directory where debugging trace files for the background processes are written during Oracle operations.

BACKUP_TAPE_IO_SLAVES = {TRUE | FALSE} [DEFERRED]

specifies whether I/O slaves are used by the Recovery Manager to back up, copy, or restore data to tape.

CONTROL_FILE_RECORD_KEEP_TIME = integer [DEFERRED]

specifies (in days) the minimum age of a record in a reusable control file section at which the record can be reused.

CREATE_STORED_OUTLINES = { TRUE | FALSE | 'category_name' } [NOOVERRIDE]

determines whether Oracle should automatically create and store an outline for each query submitted on the system. CREATE_STORED_OUTLINES is not an initialization parameter.

- TRUE enables automatic outline creation for subsequent queries in the system. These outlines receive a unique system-generated name and are stored in the DEFAULT category. If a particular query already has an outline defined for it in the DEFAULT category, that outline will remain and a new outline will not be created.
- FALSE disables automatic outline creation for the system. This is the default.
- *category_name* has the same behavior as TRUE except that any outline created in the system is stored in the *category_name* category.
- NOOVERRIDE specifies that this system setting will not override the setting for any session in which this parameter was explicitly set. If you do not specify NOOVERRIDE, this setting takes effect in all sessions.

DB_BLOCK_CHECKING = {TRUE | FALSE} DEFERRED

controls whether data block checking is done. The default is FALSE, for compatibility with earlier releases where block checking is disabled as a default.

DB_BLOCK_CHECKSUM = {TRUE | FALSE}

specifies whether the database writer background process and the direct loader will calculate a checksum and store it in the cache header of every data lock when writing to disk.

DB_BLOCK_MAX_DIRTY_TARGET = integer

limits to *integer* the number of dirty buffers in the cache and the number of buffers that will need to be read during crash or instance recovery. This parameter does **not** relate to media recovery. A value of 0 disables this parameter. The minimum accepted value to enable the parameter is 1000.

DB_FILE_MULTIBLOCK_READ_COUNT = integer

specifies the maximum number of blocks read in one I/O operation during a sequential scan.

FAST_START_IO_TARGET

specifies the target number of IOs (reads and writes) to and from buffer cache that Oracle should perform upon crash or instance recovery. Oracle continuously calculates the actual number of IOs that would be needed for recovery and compares that number against the target. If the actual number is greater than the target, Oracle attempts to write additional dirty buffers to advance the checkpoint, while minimizing the affect on performance.

For information on how to tune this parameter, see *Oracle8i Tuning*.

FAST_START_PARALLEL_ROLLBACK = { FALSE | LOW | HIGH}

specifies the number of processes spawned to perform parallel recovery.

- FALSE specifies no parallel recovery. SMON will serially recover dead transactions.
- LOW specifies that the number of recovery servers may not exceed twice the value of the CPU_COUNT parameter.
- HIGH specifies that the number of recovery servers may not exceed four times the value of the CPU_COUNT parameter.

FIXED_DATE = { 'DD_MM_YY' | 'YYYY_MI_DD_HH24_MI-SS' }

specifies a constant date for SYSDATE instead of the current date.

GC_DEFER_TIME = integer

specifies the time (in hundredths of seconds) that Oracle waits before responding to forced-write requests from other instances.

GLOBAL_NAMES = {TRUE | FALSE}

When you start an instance, Oracle determines whether to enforce global name resolution for remote objects accessed in SQL statements based on the value of the initialization parameter GLOBAL_NAMES. This system parameter enables or disables global name resolution while your instance is running. TRUE enables the enforcement of global names. FALSE disables the enforcement of global names. You can also enable or disable global name resolution for your session with the GLOBAL_NAMES parameter of the ALTER SESSION statement.

Oracle recommends that you enable global name resolution if you use or plan to use distributed processing. For more information on global name resolution and how Oracle enforces it, see "Referring to Objects in Remote Databases" on page 2-74 and *Oracle8i Distributed Database Systems*.

HASH_MULTIBLOCK_IO_COUNT = integer

specifies the number of data blocks to read and write during a hash join operation. The value multiplied by the DB_BLOCK_SIZE initialization parameter should not exceed 64 K. The default value for this parameter is 1. If the multi-threaded server is used, the value is always 1, and any value given here is ignored.

HS_AUTOREGISTER = {TRUE | FALSE}

enables or disables automatic self-registration of non-Oracle system characteristics in the Oracle server's data dictionary by Heterogeneous Services agents. For more information on accessing non-Oracle systems through Heterogeneous Services, see *Oracle8i Distributed Database Systems.*

JOB_QUEUE_PROCESSES = integer

specifies the number of job queue processes per instance (SNP*n*, where *n* is 0 to 9 followed by A to Z). Set this parameter to 1 or higher if you wish to have your snapshots updated automatically. One job queue process is usually sufficient unless you have many snapshots that refresh simultaneously.

Oracle also uses job queue processes to process requests created by the DBMS_JOB package. For more information on managing table snapshots, see *Oracle8i Replication.*

LICENSE_MAX_SESSIONS = integer

resets (for the current instance) the value of the initialization parameter LICENSE_MAX_SESSIONS, which establishes the concurrent usage licensing limit, or the limit for concurrent sessions. Once this limit is reached, only users with RESTRICTED SESSION system privilege can connect. A value of 0 disables the limit.

If you reduce the limit on sessions below the current number of sessions, Oracle does not end existing sessions to enforce the new limit. However, users without RESTRICTED SESSION system privilege can begin new sessions only when the number of sessions falls below the new limit.

Do not disable or raise session limits unless you have appropriately upgraded your Oracle license. For more information, contact your Oracle sales representative.

LICENSE_MAX_USERS = integer

resets (for the current instance) the value of the initialization parameter LICENSE_MAX_USERS, which establishes the limit for users connected to your database. Once this limit is reached, more users cannot connect. A value of 0 disables the limit.

Restriction: You cannot reduce the limit on users below the current number of users created for the database.

Do not disable or raise user limits unless you have appropriately upgraded your Oracle license. For more information, contact your Oracle sales representative.

LICENSE_SESSIONS_WARNING = integer

resets (for the current instance) the value of the initialization parameter LICENSE_SESSIONS_WARNING, which establishes a warning threshold for concurrent usage. Once this threshold is reached, Oracle writes warning messages to the database ALERT file for each subsequent session. Also, users with RESTICTED SESSION system privilege receive warning messages when they begin subsequent sessions. A value of 0 disables the warning threshold.

If you reduce the warning threshold for sessions below the current number of sessions, Oracle writes a message to the ALERT file for all subsequent sessions.

```
LOG_ARCHIVE_DEST = string
```

specifies a valid operating system pathname as the primary destination for all archive redo log file groups.

Restrictions: If you set a value for this parameter:

- You cannot have a value for LOG_ARCHIVE_DEST_*n* in your initialization parameter file, nor can you set a value for that parameter using the ALTER SESSION or ALTER SYSTEM statement.
- You cannot set a value for the parameter LOG_ARCHIVE_MIN_SUCCEED_DEST using the ALTER SESSION statement.

```
LOG_ARCHIVE_DEST_n = null_string
                     |{LOCATION=pathname | SERVICE=servicename}
                     [MANDATORY | OPTIONAL]
                     [REOPEN[=retry_time_in_seconds]]
```

specifies up to five valid operating system pathnames or Oracle service names (plus other related options) as destinations for archive redo log file groups (*n* = integers 1 through 5). For a description of the options, refer to *Oracle8i Reference*.

Restrictions: If you set a value for this parameter:

- You cannot have definitions for the parameters LOG_ARCHIVE_DEST or LOG_ARCHIVE_DUPLEX_DEST in your initialization parameter file, nor can you set values for those parameters using the ALTER SYSTEM statement.
- You cannot start archiving to a specific location using the ALTER SYSTEM ARCHIVE LOG TO *location* statement.

```
LOG_ARCHIVE_DEST_STATE_n = {ENABLE | DEFER}
```

specifies the state associated with the corresponding LOG_ARCHIVE_DEST_n parameter.

- ENABLE specifies that any associated valid destination can be used for archiving. This is the default.
- DEFER specifies that Oracle will not consider for archiving any destination associated with the corresponding LOG_ARCHIVE_DEST_n parameter.

```
LOG_ARCHIVE_DUPLEX_DEST = string
```

specifies a valid operating system pathname as the secondary destination for all archive redo log file groups.

Restriction: If you set a value for this parameter:

- You must have a definition for `LOG_ARCHIVE_DEST`.
- You cannot have a value for the parameter `LOG_ARCHIVE_DEST_n` in your initialization parameter file, nor can you set a value for that parameter using the `ALTER SYSTEM` or `ALTER SESSION` statement.
- You cannot set a value for the parameter `LOG_ARCHIVE_MIN_SUCCEED_DEST` using the `ALTER SESSION` statement.

`LOG_ARCHIVE_MAX_PROCESSES = integer`

specifies the number of archiver processes that are invoked. Permitted values are integers 1 through 10, inclusive. The default is 1.

`LOG_ARCHIVE_MIN_SUCCEED_DEST = integer`

specifies the minimum number of destinations that must succeed in order for the online log file to be available for reuse.

`LOG_CHECKPOINT_INTERVAL = integer`

limits to *integer* the number of redo blocks that can exist between an incremental checkpoint and the last block written to the redo log.

`LOG_CHECKPOINT_TIMEOUT = integer`

limits the incremental checkpoint to be at the position where the last write to the redo log (sometimes called the "tail of the log") was *integer* seconds ago, and signifies that no buffer will remain dirty (in the cache) for more than *integer* seconds. The default is 1800 seconds.

`MAX_DUMP_FILE_SIZE = { size | 'UNLIMITED'} [DEFERRED]`

specifies the trace dump file size upper limit for all user sessions. Specify the maximum *size* as either a nonnegative integer that represents the number of blocks, or as 'UNLIMITED'. If you specify 'UNLIMITED', no upper limit is imposed.

Multi-Threaded Server Parameters: When you start your instance, Oracle creates shared server processes and dispatcher processes for the multi-threaded server architecture based on the values of the MTS_SERVERS and MTS_DISPATCHERS initialization parameters. You can set the MTS_SERVERS and MTS_DISPATCHERS session parameters to perform one of the following operations while the instance is running:

- Create additional shared server processes by increasing the minimum number of shared server processes.
- Terminate existing shared server processes after their current calls finish processing.
- Create more dispatcher processes for a specific protocol, up to a maximum across all protocols specified by the initialization parameter MTS_MAX_DISPATCHERS.
- Terminate existing dispatcher processes for a specific protocol after their current user processes disconnect from the instance.

For more information on multi-threaded server architecture, see *Oracle8i Concepts*, *Oracle8i Tuning*, and *Oracle8i Parallel Server Concepts and Administration*.

```
MTS_DISPATCHERS = 'dispatch_clause'
```

dispatch_clause::=

```
(PROTOCOL = protocol) |
( ADDRESS = address) |
(DESCRIPTION = description )
[options_clause]
```

options_clause::=

```
(DISPATCHERS = integer |
 SESSIONS = integer |
 CONNECTIONS = integer |
 TICKS = seconds |
 POOL = { 1 | ON | YES | TRUE | BOTH | ({IN|OUT} = ticks) |
          0 | OFF | NO | FALSE | ticks} |
 MULTIPLEX = {1 | ON | YES | TRUE | 0 | OFF | NO |
              FALSE | BOTH | IN | OUT} |
 LISTENER = tnsname |
 SERVICE = service |
 PRESENTATION = { TTC | RO | GIOP | ejb_presentation_class } |
 INDEX = integer)
```

modifies or creates the configuration of dispatcher processes.

You can specify multiple `MTS_DISPATCHERS` parameters in a single statement for multiple network protocols. For more information on this parameter, see *Net8 Administrator's Guide* and *Oracle8i Administrator's Guide*.

```
MTS_SERVERS = integer
```

specifies a new minimum number of shared server processes.

```
QUERY_REWRITE_ENABLED = { TRUE | FALSE } [DEFERRED | NOOVERRIDE]
```

enables or disables query rewrite on all materialized views that have not been explicitly disabled. By default, TRUE enables query rewrite for all sessions immediately. Query rewrite is superseded and disabled by rule-based optimization (that is, if the OPTIMIZER_MODE parameter is set to RULE). Also enables or disables use of any function-based indexes defined on the materialized view.

- DEFERRED specifies that query rewrite is enabled or disabled only for future sessions.
- NOOVERRIDE specifies that query rewrite is enabled or disabled for all sessions that have not explicitly set this parameter using ALTER SESSION.

Note: Enabling or disabling query rewrite does not affect queries that have already been compiled, even if they are reissued. Enabling or disabling query rewrite does not affect descending indexes. For more information on query rewrite, see *Oracle8i Tuning*.

`OBJECT_CACHE_MAX_SIZE_PERCENT = integer [DEFERRED]`

specifies the percentage of the optimal cache size that the session object cache can grow past the optimal size.

`OBJECT_CACHE_OPTIMAL_SIZE = integer [DEFERRED]`

specifies (in kilobytes) the size to which the session object cache is reduced if it exceeds the maximum size.

`PARALLEL_ADAPTIVE_MULTI_USER = {TRUE | FALSE}`

specifies that Oracle should vary the degree of parallelism based on the total perceived load on the system.

`PARALLEL_INSTANCE_GROUP = 'text'`

specifies the name of the Oracle Parallel Server instance group to be used for spawning parallel query slaves.

`PARALLEL_THREADS_PER_CPU = integer`

used to compute the degree of parallelism for parallel operations where the degree of parallelism is unset. The default is operating system dependent.

`PLSQL_V2_COMPATIBILITY = {TRUE | FALSE} [DEFERRED]`

modifies the compile-time behavior of PL/SQL programs to allow language constructs that are illegal in Oracle8 and Oracle8*i* (PL/SQL V3), but were legal in Oracle7 (PL/SQL V2). See *PL/SQL User's Guide and Reference* and *Oracle8i Reference* for more information about this system parameter.

`TRUE`	enables Oracle8*i* PL/SQL V3 programs to execute Oracle7 PL/SQL V2 constructs.
`FALSE`	disallows illegal Oracle7 PL/SQL V2 constructs. This is the default.

`REMOTE_DEPENDENCIES_MODE = {TIMESTAMP | SIGNATURE}`

specifies how dependencies of remote stored procedures are handled by the server. For more information, see *Oracle8i Application Developer's Guide - Fundamentals*.

`RESOURCE_LIMIT = {TRUE | FALSE}`

When you start an instance, Oracle enforces resource limits assigned to users based on the value of the RESOURCE_LIMIT initialization parameter. This system parameter enables or disables resource limits for subsequent sessions. TRUE enables resource limits. FALSE disables resource limits.

Enabling resource limits only causes Oracle to enforce the resource limits already assigned to users. To choose resource limit values for a user, you must create a profile and assign that profile to the user. For more information, see "CREATE PROFILE" on page 7-338 and "CREATE USER" on page 7-425.

`RESOURCE_MANAGER_PLAN = plan_name`

specifies the name of the resource plan Oracle should use to allocate system resources among resource consumer groups. For information on resource consumer groups and resource plans, refer to *Oracle8i Administrator's Guide*.

`SORT_AREA_RETAINED_SIZE = integer DEFERRED`

specifies (in bytes) the maximum amount of memory that each sort operation will retain after the first fetch is done, until the cursor ends. The default is the value of the SORT_AREA_SIZE parameter.

`SORT_AREA_SIZE = integer DEFERRED`

specifies (in bytes) the maximum amount of memory to use for each sort operation. The default is operating system dependent.

`SORT_MULTIBLOCK_READ_COUNT = integer DEFERRED`

specifies the number of database blocks to read each time a sort performs a read from temporary segments. The default is 2.

`STANDBY_ARCHIVE_DEST = string`

specifies a valid operating system pathname as the standby database destination for the archive redo log files.

`TIMED_STATISTICS = {TRUE | FALSE}`

specifies whether the server requests the time from the operating system when generating time-related statistics. The default is FALSE.

`TIMED_OS_STATISTICS = integer`

specifies that operating system statistics will be collected when a request is made from a client to the server or when a request completes.

`TRANSACTION_AUDITING = {TRUE | FALSE} DEFERRED`

specifies whether the transaction layer generates a special redo record containing session and user information.

`USE_STORED_OUTLINES = { TRUE | FALSE | 'category_name' } [NOOVERRIDE]`

determines whether the optimizer will use stored outlines to generate execution plans. `USE_STORED_OUTLINES` is not an initialization parameter.

- `TRUE` causes the optimizer to use outlines stored in the DEFAULT category when compiling requests.
- `FALSE` specifies that the optimizer should not use stored outlines. This is the default.
- *category_name* causes the optimizer to use outlines stored in the *category_name* category when compiling requests.

- NOOVERRIDE specifies that this system setting will not override the setting for any session in which this parameter was explicitly set. If you do not specify NOOVERRIDE, this setting takes effect in all sessions.

USER_DUMP_DEST = 'directory_name'

specifies the pathname where Oracle will write debugging trace files on behalf of a user process.

Examples

Archive Log Examples The following statement manually archives the redo log file group with the log sequence number 4 in thread number 3:

```
ALTER SYSTEM ARCHIVE LOG THREAD 3 SEQUENCE 4;
```

The following statement manually archives the redo log file group containing the redo log entry with the SCN 9356083:

```
ALTER SYSTEM ARCHIVE LOG CHANGE 9356083;
```

The following statement manually archives the redo log file group containing a member named 'DISKL:LOG6.LOG' to an archived redo log file in the location 'DISKA:[ARCH$]':

```
ALTER SYSTEM ARCHIVE LOG
    LOGFILE 'diskl:log6.log'
    TO 'diska:[arch$]';
```

Query Rewrite Example This statement enables query rewrite in all sessions for all materialized views that have not been explicitly disabled:

```
ALTER SYSTEM SET QUERY_REWRITE_ENABLED = TRUE;
```

Restricted Session Example You may want to restrict logons if you are performing application maintenance and you want only application developers with RESTRICTED SESSION system privilege to log on. To restrict logons, issue the following statement:

```
ALTER SYSTEM ENABLE RESTRICTED SESSION;
```

You can then terminate any existing sessions using the KILL SESSION clause of the ALTER SYSTEM statement.

After performing maintenance on your application, issue the following statement to allow any user with CREATE SESSION system privilege to log on:

```
ALTER SYSTEM DISABLE RESTRICTED SESSION;
```

Shared Pool Example You might want to clear the shared pool before beginning performance analysis. To clear the shared pool, issue the following statement:

```
ALTER SYSTEM FLUSH SHARED_POOL;
```

CHECKPOINT Example The following statement forces a checkpoint:

```
ALTER SYSTEM CHECKPOINT;
```

Resource Limit Example This ALTER SYSTEM statement dynamically enables resource limits:

```
ALTER SYSTEM SET RESOURCE_LIMIT = TRUE;
```

Multi-Threaded Server Examples The following statement changes the minimum number of shared server processes to 25:

```
ALTER SYSTEM SET MTS_SERVERS = 25;
```

If there are currently fewer than 25 shared server processes, Oracle creates more. If there are currently more than 25, Oracle terminates some of them when they are finished processing their current calls if the load could be managed by the remaining 25.

The following statement dynamically changes the number of dispatcher processes for the TCP/IP protocol to 5 and the number of dispatcher processes for the DECNET protocol to 10:

```
ALTER SYSTEM
   SET MTS_DISPATCHERS =
      '(INDEX=0)(PROTOCOL=TCP)(DISPATCHERS=5)',
      '(INDEX=1)(PROTOCOL=DECNet)(DISPATCHERS=10)';
```

If there are currently fewer than 5 dispatcher processes for TCP, Oracle creates new ones. If there are currently more than 5, Oracle terminates some of them after the connected users disconnect.

If there are currently fewer than 10 dispatcher processes for DECnet, Oracle creates new ones. If there are currently more than 10, Oracle terminates some of them after the connected users disconnect.

If there are currently existing dispatchers for another protocol, the above statement does not affect the number of dispatchers for that protocol.

Licensing Examples The following statement dynamically changes the limit on sessions for your instance to 64 and the warning threshold for sessions on your instance to 54:

```
ALTER SYSTEM
   SET LICENSE_MAX_SESSIONS = 64
   LICENSE_SESSIONS_WARNING = 54;
```

If the number of sessions reaches 54, Oracle writes a warning message to the ALERT file for each subsequent session. Also, users with RESTRICTED SESSION system privilege receive warning messages when they begin subsequent sessions.

If the number of sessions reaches 64, only users with RESTRICTED SESSION system privilege can begin new sessions until the number of sessions falls below 64 again.

The following statement dynamically disables the limit for sessions on your instance. After you issue the above statement, Oracle no longer limits the number of sessions on your instance.

```
ALTER SYSTEM SET LICENSE_MAX_SESSIONS = 0;
```

The following statement dynamically changes the limit on the number of users in the database to 200. After you issue the above statement, Oracle prevents the number of users in the database from exceeding 200.

```
ALTER SYSTEM SET LICENSE_MAX_USERS = 200;
```

SWITCH LOGFILE Example You may want to force a log switch to drop or rename the current redo log file group or one of its members, because you cannot drop or rename a file while Oracle is writing to it. The forced log switch affects only your instance's redo log thread. The following statement forces a log switch:

```
ALTER SYSTEM
   SWITCH LOGFILE;
```

Distributed Recovery Example The following statement enables distributed recovery:

```
ALTER SYSTEM ENABLE DISTRIBUTED RECOVERY;
```

You may want to disable distributed recovery for demonstration or testing purposes.You can disable distributed recovery in both single-process and multiprocess mode with the following statement:

```
ALTER SYSTEM DISABLE DISTRIBUTED RECOVERY;
```

When your demonstration or testing are complete, you can then enable distributed recovery again by issuing an ALTER SYSTEM statement with the ENABLE DISTRIBUTED RECOVERY clause.

KILL SESSION Example You may want to kill the session of a user that is holding resources needed by other users. The user receives an error message indicating that the session has been killed. That user can no longer make calls to the database without beginning a new session. Consider this data from the V$SESSION dynamic performance table:

```
SELECT sid, serial, username
FROM v$session

  SID    SERIAL USERNAME
----- --------- ----------------
    1         1
    2         1
    3         1
    4         1
    5         1
    7         1
    8        28 OPS$BQUIGLEY
   10       211 OPS$SWIFT
   11        39 OPS$OBRIEN
   12        13 SYSTEM
   13         8 SCOTT
```

The following statement kills the session of the user SCOTT using the SID and SERIAL# values from V$SESSION:

```
ALTER SYSTEM KILL SESSION '13, 8';
```

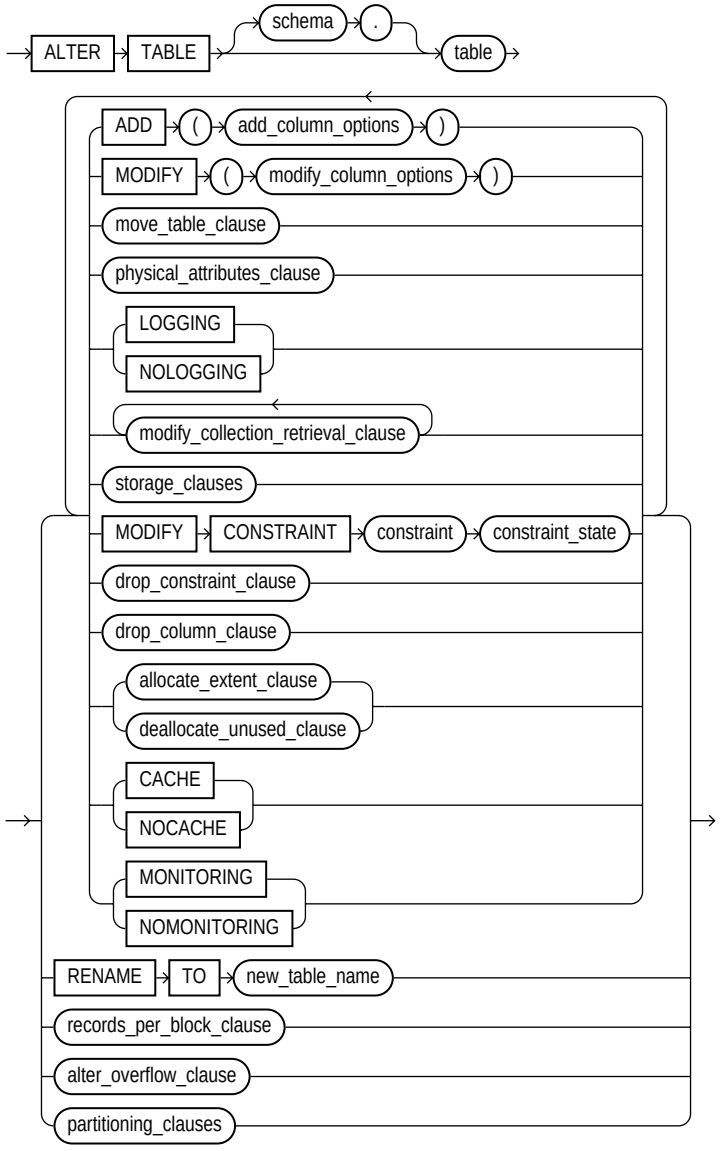
DISCONNECT SESSION Example The following statement disconnects user SCOTT's session, using the SID and SERIAL# values from V$SESSION:

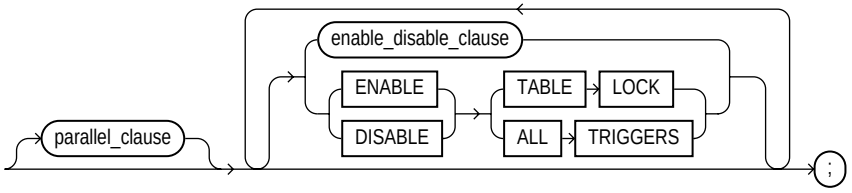
```
ALTER SYSTEM DISCONNECT SESSION '13, 8' POST_TRANSACTION;
```

For more information about application failover, see *Oracle8i Parallel Server Concepts and Administration* and *Oracle8i Tuning.*

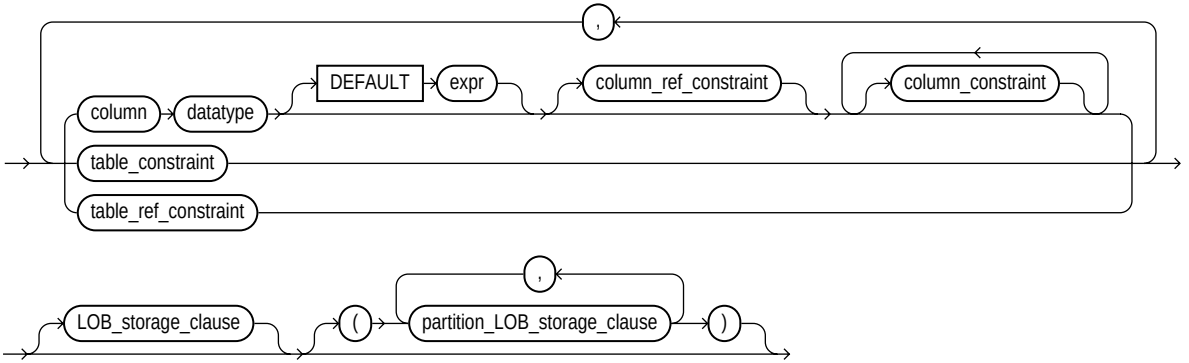
ALTER TABLE

Syntax



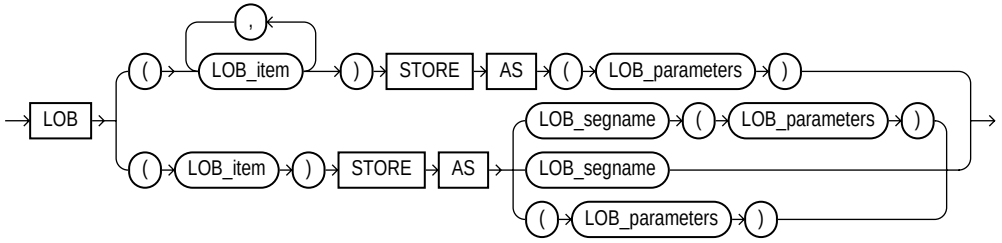


add_column_options::=

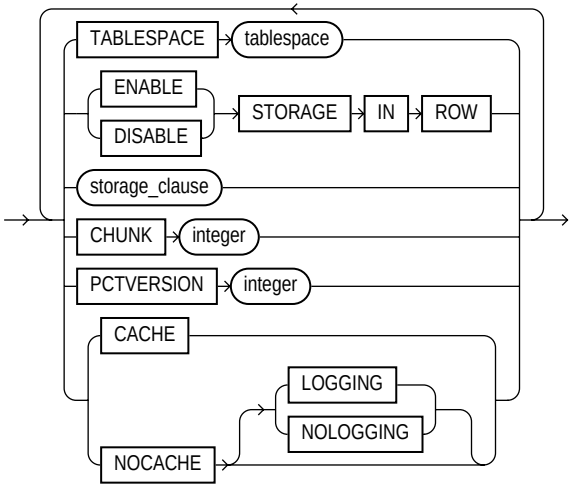


column_constraint, table_constraint, column_ref_constraint, table_ref_constraint, constraint_state: See the "constraint_clause" on page 7-217.

LOB_storage_clause::=

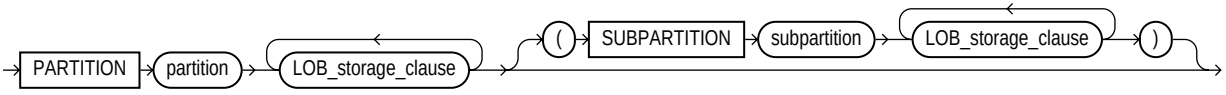


LOB_parameters::=

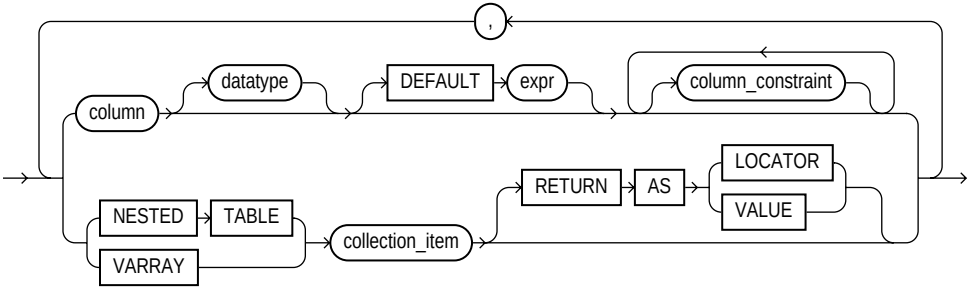


storage_clause: See "storage_clause" on page 7-575.

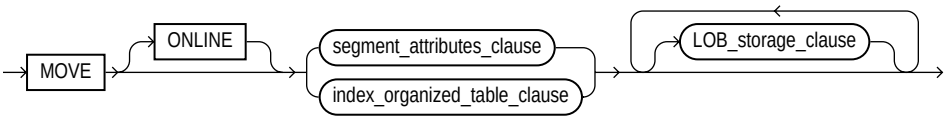
partition_LOB_storage_clause::=



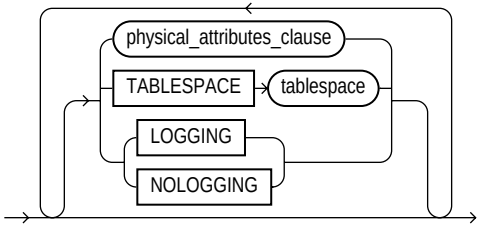
modify_column_options::=



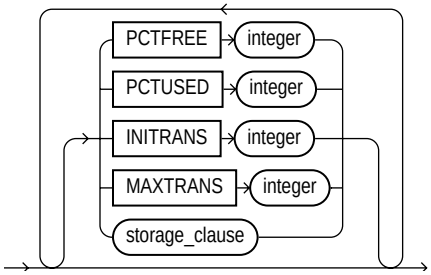
move_table_clause::=



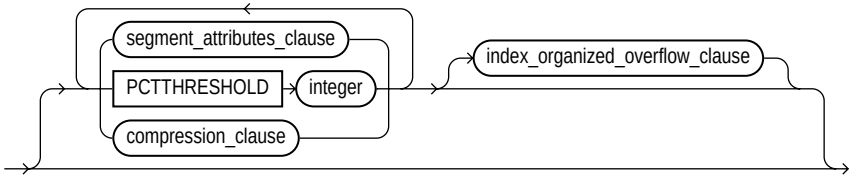
segment_attributes_clause::=



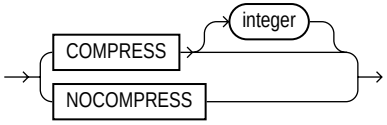
physical_attributes_clause::=



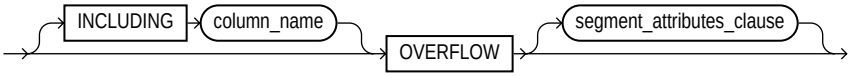
index_organized_table_clause::=



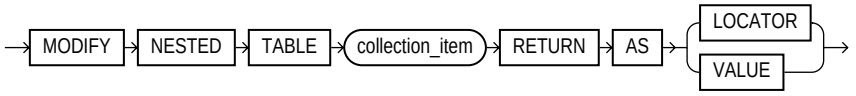
compression_clause::=



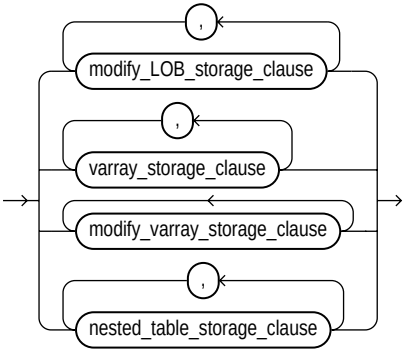
index_organized_overflow_clause::=



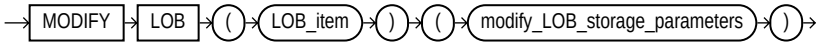
modify_collection_retrieval_clause::=



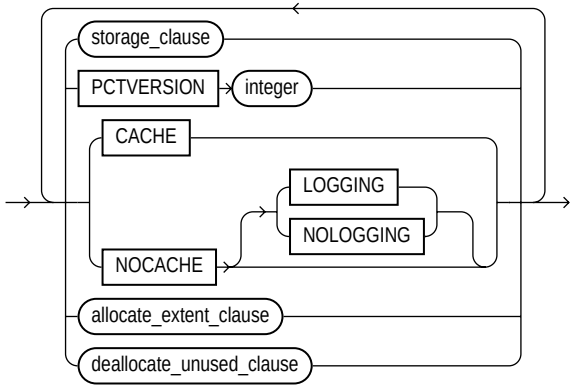
storage_clauses::=



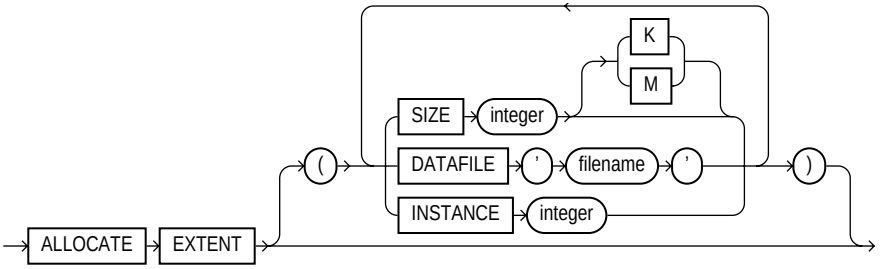
modify_LOB_storage_clause::=



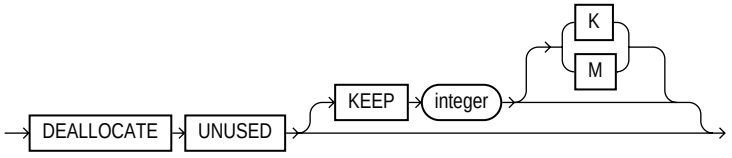
modify_LOB_storage_parameters::=



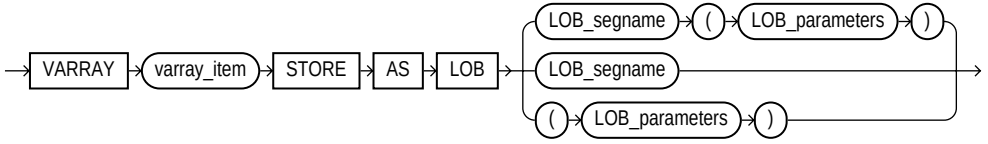
allocate_extent_clause::=



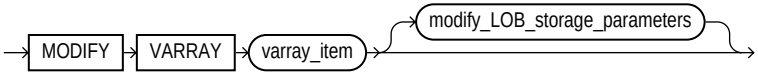
deallocate_unused_clause::=



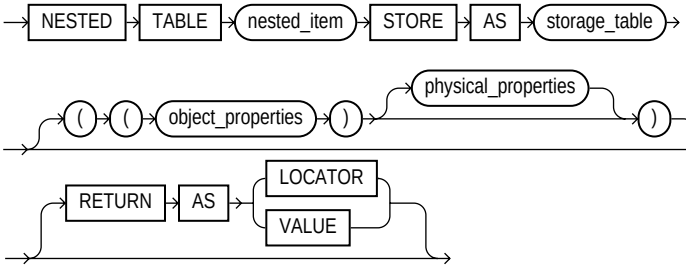
varray_storage_clause::=



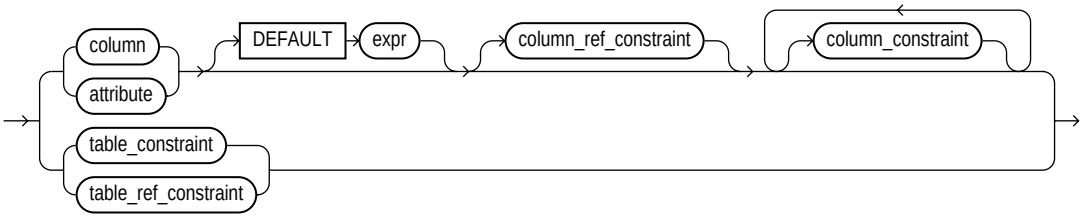
modify_varray_storage_clause::=



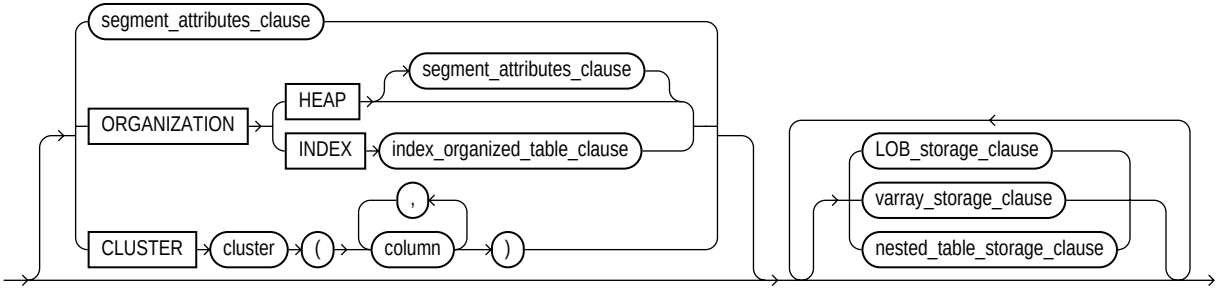
nested_table_storage_clause::=



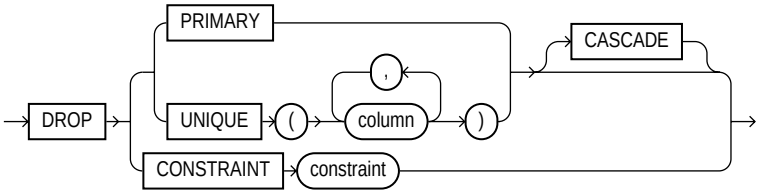
object_properties::=



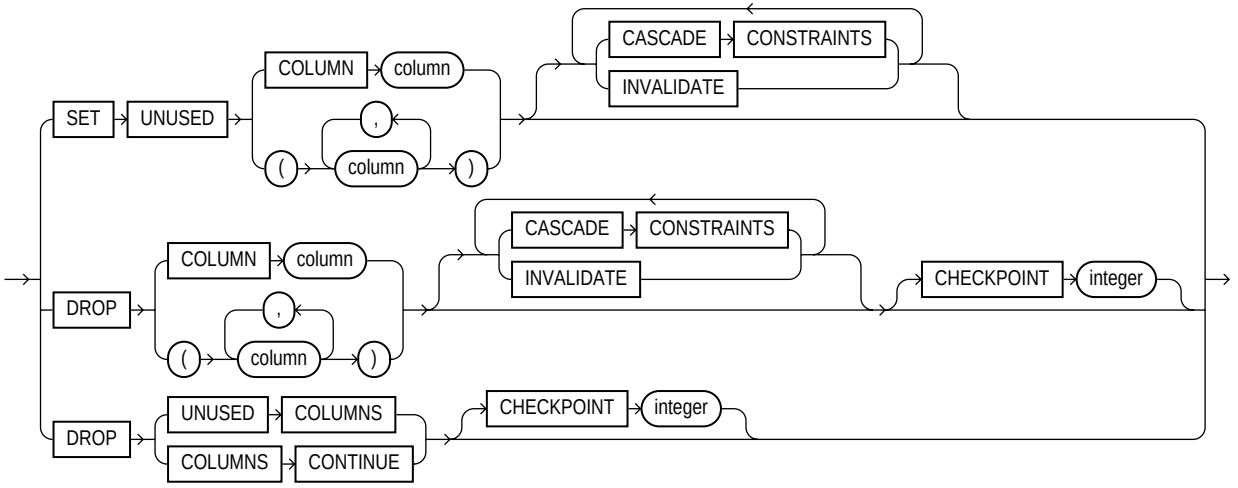
physical_properties::=



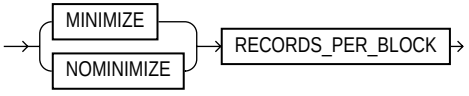
drop_constraint_clause::=



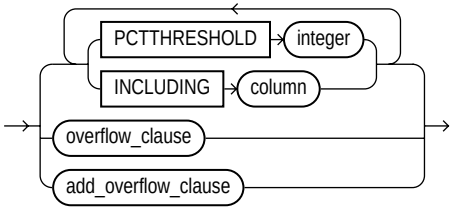
drop_column_clause::=



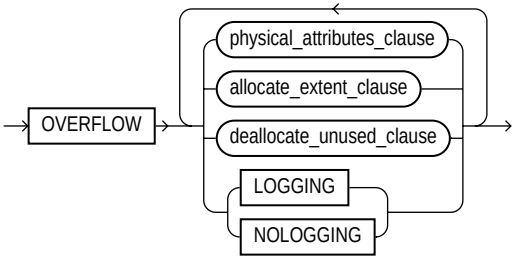
records_per_block_clause::=



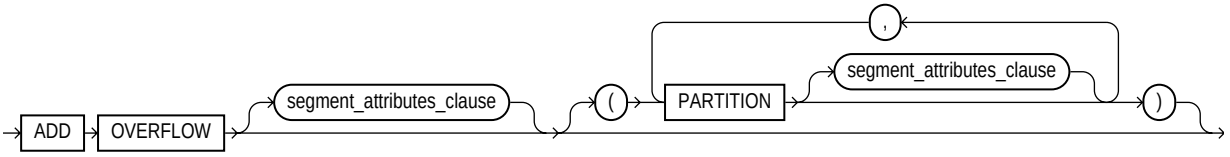
alter_overflow_clause::=



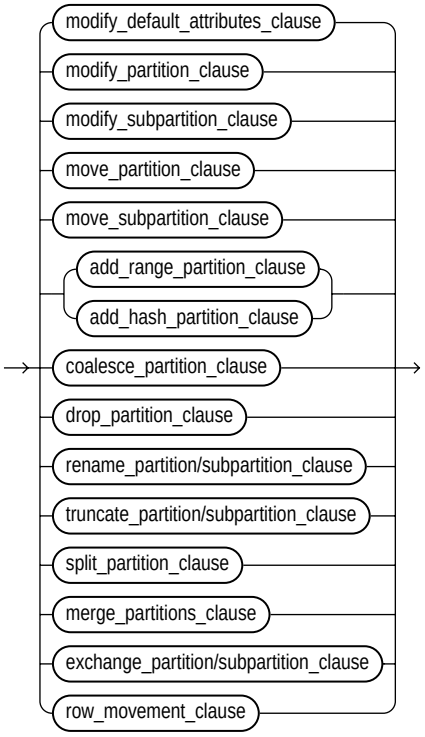
overflow_clause::=



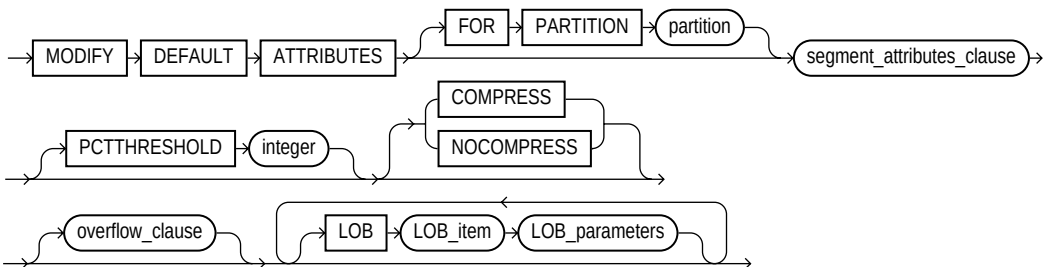
add_overflow_clause::=



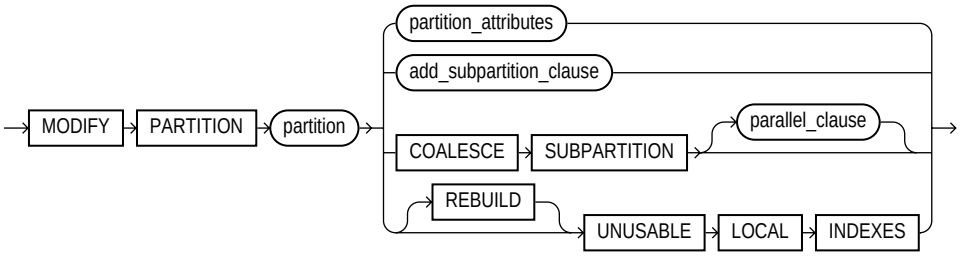
partitioning_clauses::=



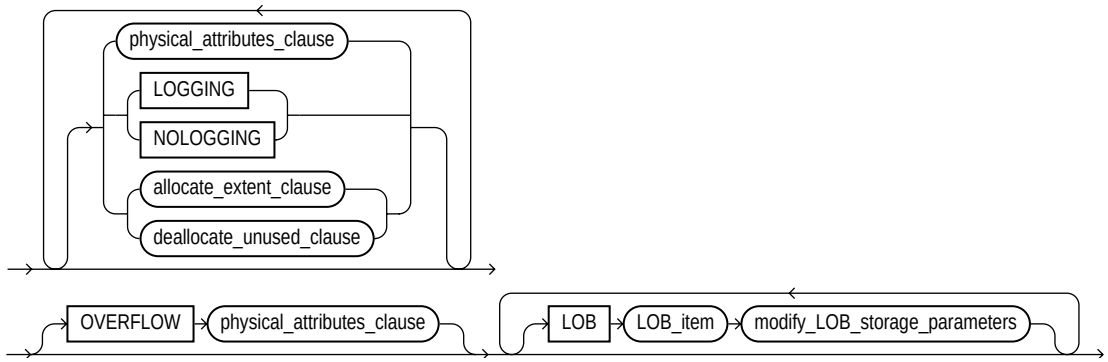
modify_default_attributes_clause::=



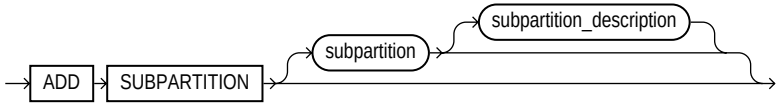
modify_partition_clause::=



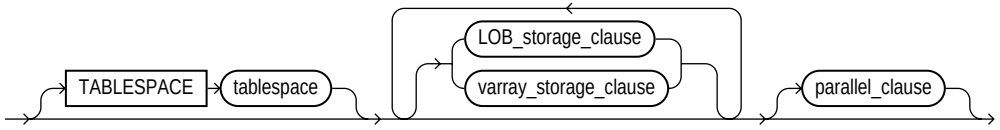
partition_attributes::=



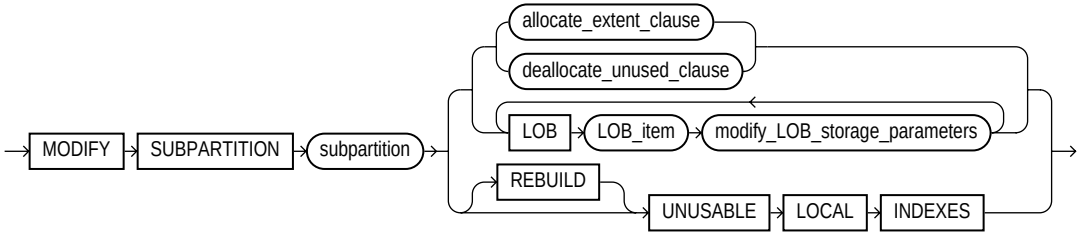
add_subpartition_clause::=



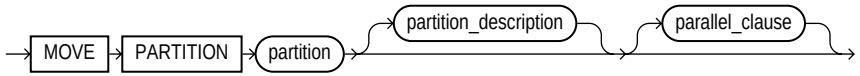
subpartition_description::=



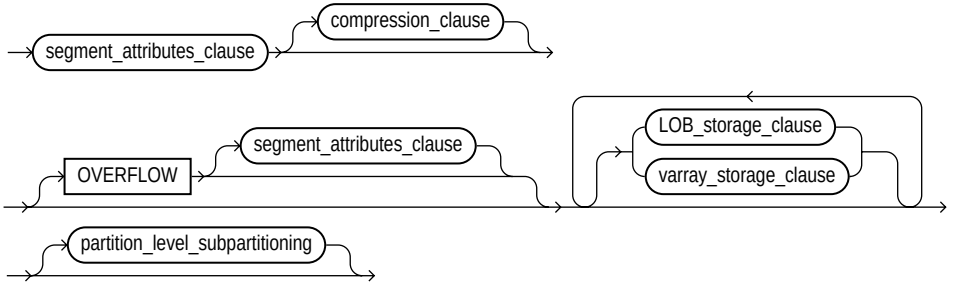
modify_subpartition_clause::=



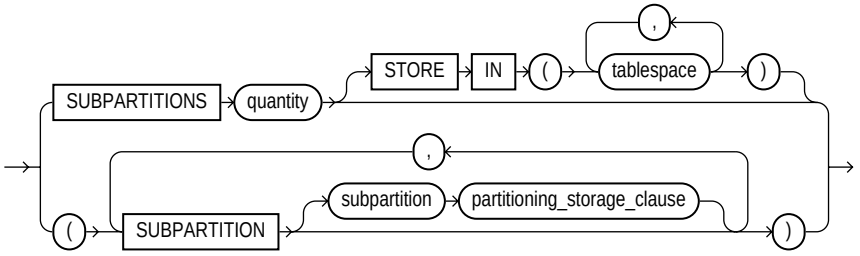
move_partition_clause::=



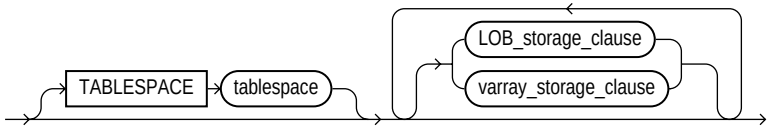
partition_description::=



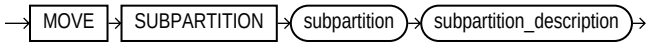
partition_level_subpartitioning::=



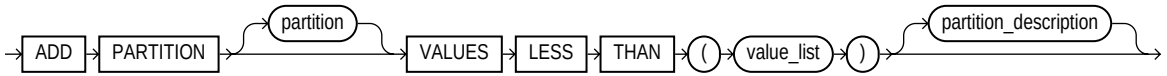
partitioning_storage_clause::=



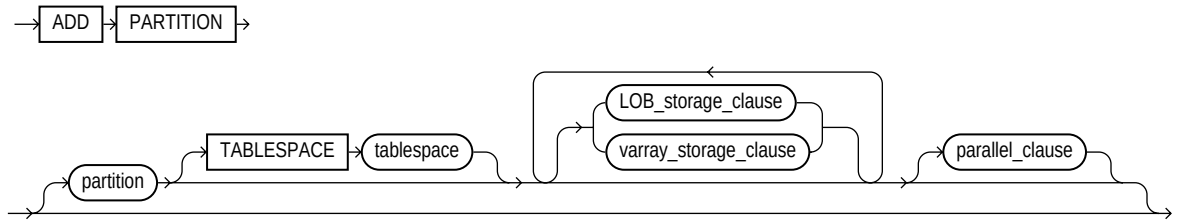
move_subpartition_clause::=



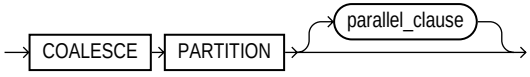
add_range_partition_clause::=



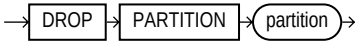
add_hash_partition_clause::=



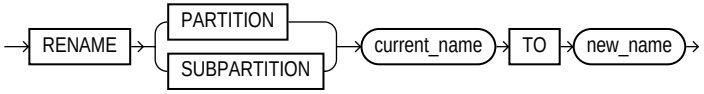
coalesce_partition_clause::=



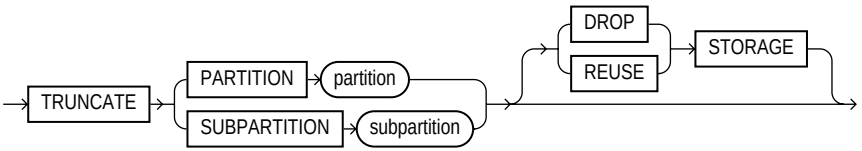
drop_partition_clause::=



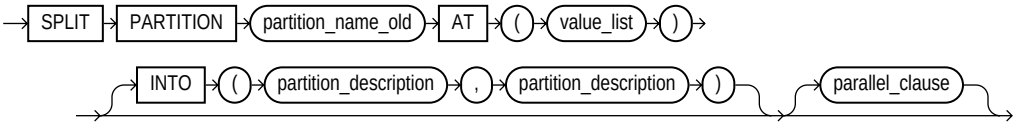
rename_partition/ subpartition_clause::=



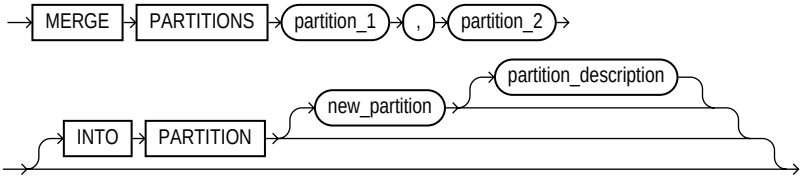
truncate_partition_clause/truncate_partition_clause::=



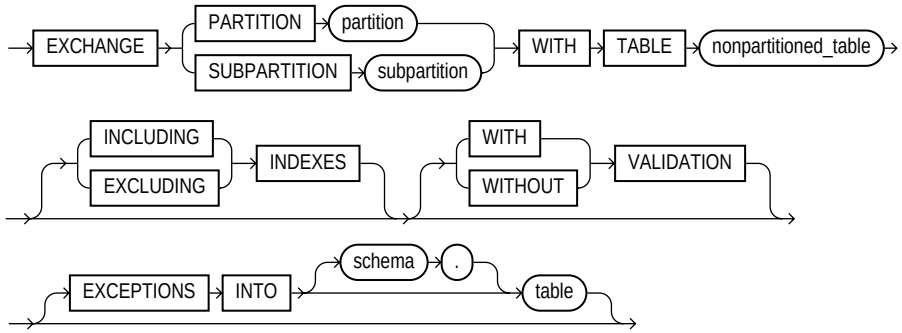
split_partition_clause::=



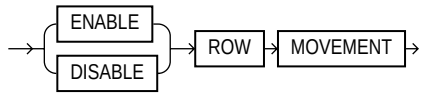
merge_partitions_clause::=



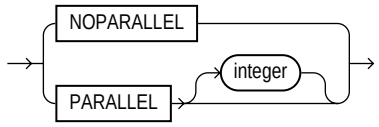
exchange_partition_clause/exchange_partition_clause::=



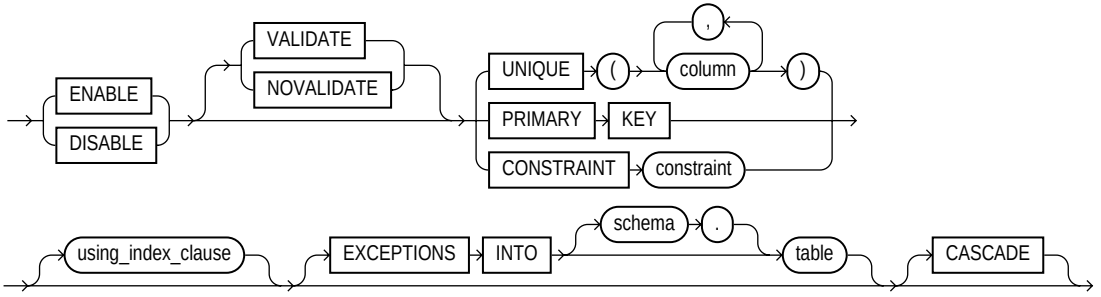
row_movement_clause::=



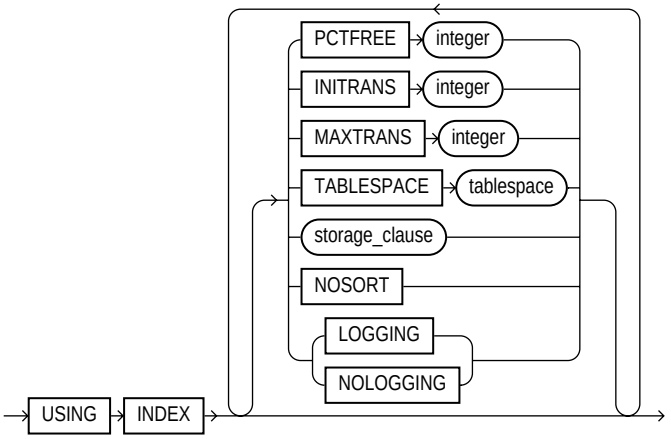
parallel_clause::=



enable_disable_clause::=



using_index_clause::=



Purpose

To alter the definition of a nonpartitioned table, a partitioned table, a table partition, or a table subpartition.

Prerequisites

The table must be in your own schema, or you must have ALTER privilege on the table, or you must have ALTER ANY TABLE system privilege. For some operations you may also need the CREATE ANY INDEX privilege.

In addition, if you are not the owner of the table, you need the DROP ANY TABLE privilege in order to use the *drop_partition_clause* or *truncate_partition_clause*.

You must also have space quota in the tablespace in which space is to be acquired in order to use the *add_partition_clause*, *modify_partition_clause*, *move_partition_clause*, and *split_partition_clause*.

To enable a UNIQUE or PRIMARY KEY constraint, you must have the privileges necessary to create an index on the table. You need these privileges because Oracle creates an index on the columns of the unique or primary key in the schema containing the table. See "CREATE INDEX" on page 7-273.

To enable or disable triggers, the triggers must be in your schema or you must have the ALTER ANY TRIGGER system privilege.

To use an object type in a column definition when modifying a table, either that object must belong to the same schema as the table being altered, or you must have either the EXECUTE ANY TYPE system privilege or the EXECUTE schema object privilege for the object type.

Keywords and Parameters

The clauses described below have specialized meaning in the ALTER TABLE statement. For descriptions of the remaining keywords, see "CREATE TABLE" on page 7-359.

Note: Operations performed by the ALTER TABLE statement can cause Oracle to invalidate procedures and stored functions that access the table. For information on how and when Oracle invalidates such objects, see *Oracle8i Concepts*.

schema is the schema containing the table. If you omit *schema*, Oracle assumes the table is in your own schema.

table is the name of the table to be altered.

You can modify, or drop columns from, or rename a temporary table. However, for a temporary table, you cannot:

- Add columns of nested-table or varray type. You can add columns of other types.
- Specify referential integrity (foreign key) constraints for an added or modified column
- Specify the following clauses of the *LOB_storage_clause* for an added or modified LOB column: TABLESPACE, *storage_clause*, LOGGING | NOLOGGING, or the *LOB_index_clause*.
- Specify the *physical_attribute_clause*, *nested_table_storage_clause*, *parallel_clause*, *allocate_extent_clause*, *deallocate_unused_clause*, or any of the *index-organized table* clauses
- Exchange partitions between a partition and a temporary table
- Specify LOGGING or NOLOGGING
- Specify MOVE

Note: If you alter a table that is a master table for one or more materialized views, the materialized views are marked INVALID. Invalid materialized views cannot be used by query rewrite and cannot be refreshed. To revalidate a materialized view, see "ALTER MATERIALIZED VIEW / SNAPSHOT" on page 7-45. For more information on materialized views in general, see *Oracle8i Tuning*.

ADD

add_column_options

adds a column or integrity constraint. If you add a column, the initial value of each row for the new column is null. For a description of the keywords and parameters of this clause, see "CREATE TABLE" on page 7-359.

You can add an overflow data segment to each partition of a partitioned index-organized table.

You can add LOB columns to nonpartitioned and partitioned tables. You can specify LOB storage at the table and at the partition or subpartition level.

If you previously created a view with a query that used the "SELECT *" syntax to select all columns from *table*, and you now add a column to *table*, Oracle does not automatically add the new column to the view. To add the new column to the view, re-create the view using the CREATE VIEW statement with the OR REPLACE clause. See "CREATE VIEW" on page 7-430.

Restrictions:

- You cannot add a LOB column to a partitioned index-organized table. (This restriction does not apply to nonpartitioned index-organized tables.)
- You cannot add a column with a NOT NULL constraint if *table* has any rows.
- If you specify this clause for an index-organized table, you cannot specify any other clauses in the same statement.

table_ref_constraint

column_ref_constraint

These clauses let you further describe a column of type REF. The only difference between these clauses is that you specify *table_ref* from the table level, so you must identify the REF column or attribute you are defining. You specify *column_ref* after you have already identified the REF column or attribute.

For syntax and description of these constraints, including restrictions, see the "constraint_clause" on page 7-217.

column_constraint

adds or removes a NOT NULL constraint to or from an existing column. You cannot use this clause to modify any other type of constraint using ALTER TABLE. See the "constraint_clause" on page 7-217.

table_constraint

adds or modifies an integrity constraint on the table. See the "constraint_clause" on page 7-217.

LOB_storage_clause

specifies the LOB storage characteristics for the newly added LOB column. You cannot use this clause to modify an existing LOB column. Instead, you must use the *modify_LOB_storage_clause*.

lob_item

is the LOB column name or LOB object attribute for which you are explicitly defining tablespace and storage characteristics that are different from those of the table.

lob_segname

specifies the name of the LOB data segment. You cannot use *lob_segname* if more than one *lob_item* is specified.

ENABLE | DISABLE STORAGE IN ROW	specifies whether the LOB value is stored in the row (inline) or outside of the row. (The LOB locator is always stored in the row regardless of where the LOB value is stored.) ■ ENABLE specifies that the LOB value is stored inline if its length is less than approximately 4000 bytes minus system control information. This is the default. ■ DISABLE specifies that the LOB value is stored outside of the row regardless of the length of the LOB value. **Restriction:** You cannot change STORAGE IN ROW once it is set. Therefore, you can specify this clause only as part of the *add_column_options* clause, not as part of the *modify_column_options* clause.
CHUNK *integer*	specifies the number of bytes to be allocated for LOB manipulation. If *integer* is not a multiple of the database block size, Oracle rounds up (in bytes) to the next multiple. For example, if the database block size is 2048 and *integer* is 2050, Oracle allocates 4096 bytes (2 blocks).The maximum value is 32768 (32 K), which is the largest Oracle block size allowed. The default CHUNK size is one Oracle database block. You cannot change the value of CHUNK once it is set. **Note:** The value of CHUNK must be less than or equal to the value of NEXT (either the default value or that specified in the storage clause). If CHUNK exceeds the value of NEXT, Oracle returns an error.
PCTVERSION *integer*	is the maximum percentage of overall LOB storage space used for creating new versions of the LOB. The default value is 10, meaning that older versions of the LOB data are not overwritten until 10% of the overall LOB storage space is used.
LOB_index_clause	This clause is deprecated as of Oracle8*i*. Oracle generates an index for each LOB column. The LOB indexes are system named and system managed, and reside in the same tablespace as the LOB data segments. Although it is still possible for you to specify this clause, Oracle Corporation strongly recommends that you no longer do so. For information on how Oracle manages LOB indexes in tables migrated from earlier versions, see *Oracle8i Migration.*

partition_LOB_storage_clause lets you specify a separate *LOB_storage_clause* for each partition. You must specify the partitions in the order of partition position.

If you do not specify a *LOB_storage_clause* for a particular partition, the storage characteristics are those specified for the LOB item at the table level. If you also did not specify any storage characteristics at the table level for the LOB item, Oracle stores the LOB data partition in the same tablespace as the table partition to which it corresponds.

Restriction: You can specify only one list of *partition_LOB_storage_clauses* per ALTER TABLE statement, and all *LOB_storage_clauses* must precede the list of *partition_LOB_storage_clause*s.

MODIFY

modify_column_options

modifies the definition of an existing column. If you omit any of the optional parts of the column definition (datatype, default value, or column constraint), these parts remain unchanged.

- You can change a CHAR column to VARCHAR2 (or VARCHAR) and a VARCHAR2 (or VARCHAR) to CHAR only if the column contains nulls in all rows or if you do not attempt to change the column size.
- You can change any column's datatype or decrease any column's size if all rows for the column contain nulls.
- You can always increase the size of a character or raw column or the precision of a numeric column, whether or not all the columns contain nulls.

Restrictions:

- You cannot modify the datatype or length of a column that is part of a table or index partitioning or subpartitioning key.
- You cannot modify the definition of a column on which a domain index has been built.
- If you specify this clause for an index-organized table, you cannot specify any other clauses in the same statement.

column is the name of the column to be added or modified.

The only type of integrity constraint that you can add to an existing column using the MODIFY clause with the column constraint syntax is a NOT NULL constraint, and only if the column contains no nulls. To define other types of integrity constraints (UNIQUE, PRIMARY KEY, referential integrity, and CHECK constraints) on existing columns, using the ADD clause and the table constraint syntax.

datatype specifies a new datatype for an existing column.

You can omit the datatype only if the statement also designates the column as part of the foreign key of a referential integrity constraint. Oracle automatically assigns the column the same datatype as the corresponding column of the referenced key of the referential integrity constraint.

If you change the datatype of a column in a materialized view container table, the corresponding materialized view is invalidated. To revalidate a materialized view, see "ALTER MATERIALIZED VIEW / SNAPSHOT" on page 7-45.

Restrictions:

- You cannot specify a column of datatype ROWID for an index-organized table, but you can specify a column of type UROWID.
- You cannot change a column's datatype to LOB or REF.

DEFAULT — specifies a new default for an existing column. Oracle assigns this value to the column if a subsequent INSERT statement omits a value for the column. If you are adding a new column to the table and specify the default value, Oracle inserts the default column value into all rows of the table.

The datatype of the default value must match the datatype specified for the column. The column must also be long enough to hold the default value. A DEFAULT expression cannot contain references to other columns, the pseudocolumns CURRVAL, NEXTVAL, LEVEL, and ROWNUM, or date constants that are not fully specified.

MODIFY CONSTRAINT *constraint* — modifies the state of an existing constraint named *constraint*. For a description of all the keywords and parameters of *constraint_state*, see the "constraint_clause" on page 7-217.

move_table_clause — For a **heap-organized table**, use the *segment_attributes_clause* of the syntax. The *move_table_clause* lets you relocate data of a nonpartitioned table into a new segment, optionally in a different tablespace, and optionally modify any of its storage attributes.

You can also move any LOB data segments associated with the table using the *LOB_storage_clause*. (LOB items not specified in this clause are not moved.)

For an **index-organized table**, use the *index_organized_table_clause* of the syntax. The *move_table_clause* rebuilds the index-organized table's primary key index B*-tree. The overflow data segment is not rebuilt unless the OVERFLOW keyword is explicitly stated, with two exceptions:

- If you alter the values of PCTTHRESHOLD or the INCLUDING column as part of this ALTER TABLE statement, the overflow data segment is rebuilt.
- If any of out-of-line columns (LOBs, varrays, nested table columns) in the index-organized table are moved explicitly, then the overflow data segment is also rebuilt.

The index and data segments of LOB columns are not rebuilt unless you specify the LOB columns explicitly as part of this ALTER TABLE statement.

ONLINE — specifies that DML operations on the index-organized table are allowed during rebuilding of the table's primary key index B*-tree.

Restrictions:

- You can specify this clause only for a nonpartitioned index-organized table.
- Parallel DML is not supported during online MOVE. If you specify ONLINE and then issue parallel DML statements, Oracle returns an error.

compression_clause — enables and disables key compression in an index-organized table.

- COMPRESS enables key compression, which eliminates repeated occurrence of primary key column values in index-organized tables. Use *integer* to specify the prefix length (number of prefix columns to compress).

  The valid range of prefix length values is from 1 to the number of primary key columns minus 1. The default prefix length is the number of primary key columns minus 1.

  Restrictions:

  - You can specify this clause only for an index-organized table.

  - You can specify compression for a partition of an index-organized table only if compression has been specified at the table level.

- NOCOMPRESS disables key compression in index-organized tables. This is the default.

TABLESPACE specifies the tablespace into which the rebuilt index-organized table is stored.

Restrictions:

- If you specify MOVE, it must be the first clause. For an index-organized table, the only clauses outside this clause that are allowed are the *physical_attribute_clause* and the *parallel_clause*. For heap-organized tables, you can specify those two clauses and the *LOB_storage_clauses*.
- You cannot MOVE an entire partitioned table (either heap or index organized). You must move individual partitions or subpartitions. See "move_partition_clause" on page 7-146 and "move_subpartition_clause" on page 7-147.

Notes regarding LOBs:

For any LOB columns you specify in this clause:

- Oracle drops the old LOB data segment and corresponding index segment and creates new segments, even if you do not specify a new tablespace.
- If the LOB index in *table* resided in a different tablespace from the LOB data, Oracle collocates the LOB index with the LOB data in the LOB data's tablespace after the move.

physical_attributes_clause changes the value of PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and storage characteristics. See the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters of "CREATE TABLE" on page 7-359 and the "storage_clause".

Restriction: You cannot specify the PCTUSED parameter for the index segment of an index-organized table.

WARNING:

- For a nonpartitioned table, the values you specify **override any values specified for the table at create time**.
- For a range- or hash-partitioned table, the values you specify are the default values for the table and the **actual values for every existing partition, overriding any values already set for the partitions**. To change default table attributes without overriding existing partition values, use the *modify_default_attributes_clause*.
- For a composite-partitioned table, the values you specify are the default values for the table and all partitions of the table and the **actual values for all subpartitions of the table, overriding any values already set for the subpartitions**. To change default partition attributes without overriding existing subpartition values, use the *modify_default_attributes_clause* with the FOR PARTITION clause.

modify_collection_retrieval_clause changes what is returned when a collection item is retrieved from the database.

collection_item is the name of a column-qualified attribute whose type is nested table or varray.

RETURN AS specifies what Oracle returns as the result of a query.

- LOCATOR specifies that a unique locator for the nested table is returned.
- VALUE specifies that a copy of the nested table itself is returned.

storage_clauses:

modify_LOB_storage_clause modifies the physical attributes of the LOB *lob_item*. You can specify only one *lob_item* for each *modify_LOB_storage_clause*.

Restriction: You cannot modify the value of the INITIAL parameter in the *storage_clause* when modifying the LOB storage attributes.

varray_storage_clause lets you specify separate storage characteristics for the LOB in which a varray will be stored. In addition, if you specify this clause, Oracle will always store the varray in a LOB, even if it is small enough to be stored inline.

Restriction: You cannot specify the TABLESPACE clause of *LOB_parameters* as part of this clause. The LOB tablespace for a varray defaults to the containing table's tablespace.

modify_varray_storage_clause lets you change the storage characteristics of an existing LOB in which a varray is stored.

Restriction: You cannot specify the TABLESPACE clause of *LOB_parameters* as part of this clause. The LOB tablespace for a varray defaults to the containing table's tablespace.

nested_table_storage_clause enables you to specify separate storage characteristics for a nested table, which in turn enables you to define the nested table as an index-organized table. You must include this clause when creating a table with columns or column attributes whose type is a nested table. (Clauses within this clause that function the same way they function for parent object tables are not repeated here.)

Restrictions:

- You cannot specify the *parallel_clause*.
- You cannot specify TABLESPACE (as part of the *segment_attributes_clause*) for a nested table. The tablespace is always that of the parent table.

nested_item is the name of a column (or a top-level attribute of the table's object type) whose type is a nested table.

storage_table is the name of the table where the rows of *nested_item* reside. The storage table is created in the same schema and the same tablespace as the parent table.

drop_constraint_clause drops an integrity constraint from the database. Oracle stops enforcing the constraint and removes it from the data dictionary. You can specify only one constraint for each *drop_constraint_clause*, but you can specify multiple *drop_constraint_clauses* in one statement.

PRIMARY KEY drops the table's PRIMARY KEY constraint.

UNIQUE drops the UNIQUE constraint on the specified columns.

CONSTRAINT drops the integrity constraint named *constraint*.

CASCADE drops all other integrity constraints that depend on the dropped integrity constraint.

Restrictions:

- You cannot drop a UNIQUE or PRIMARY KEY constraint that is part of a referential integrity constraint without also dropping the foreign key. To drop the referenced key and the foreign key together, use the CASCADE clause. If you omit CASCADE, Oracle does not drop the PRIMARY KEY or UNIQUE constraint if any foreign key references it.
- You cannot drop a primary key constraint (even with the CASCADE clause) on a table that uses the primary key as its object identifier (OID).
- If you drop a referential integrity constraint on a REF column, the REF column remains scoped to the referenced table.
- You cannot drop the scope of the column.

drop_column_clause lets you free space in the database by dropping columns you no longer need, or by marking them to be dropped at a future time when the demand on system resources is less.

SET UNUSED marks one or more columns as unused. Specifying this clause does not actually remove the target columns from each row in the table (that is, it does not restore the disk space used by these columns). Therefore, the response time is faster than it would be if you execute the DROP clause.

You can view all tables with columns marked as unused in the data dictionary views USER_UNUSED_COL_TABS, DBA_UNUSED_COL_TABS, and ALL_UNUSED_COL_TABS. For information on these views, see *Oracle8i Reference*.

Unused columns are treated as if they were dropped, even though their column data remains in the table's rows. After a column has been marked as unused, you have no access to that column. A "SELECT *" query will not retrieve data from unused columns. In addition, the names and types of columns marked unused will not be displayed during a DESCRIBE, and you can add to the table a new column with the same name as an unused column.

Note: Until you actually drop these columns, they continue to count toward the absolute limit of 1000 columns per table. (For more information, see "CREATE TABLE" on page 7-359.) Also, if you mark a column of datatype LONG as UNUSED, you cannot add another LONG column to the table until you actually drop the unused LONG column.

DROP

removes the column descriptor and the data associated with the target column from each row in the table. If you explicitly drop a particular column, all columns currently marked as unused in the target table are dropped at the same time.

When the column data is dropped:

- All indexes defined on any of the target columns are also dropped.
- All constraints that reference a target column are removed.
- If any statistics types are associated with the target columns, Oracle disassociates the statistics from the column with the FORCE option and drops any statistics collected using the statistics type. For more information on disassociating statistics types, see "DISASSOCIATE STATISTICS" on page 7-444.

Note: If a constraint also references a nontarget column, Oracle returns an error and does not drop the column unless you have specified the CASCADE CONSTRAINTS clause. If you have specified that clause, Oracle removes all constraints that reference any of the target columns.

DROP UNUSED COLUMNS

removes from the table all columns currently marked as unused. Use this statement when you want to reclaim the extra disk space from unused columns in the table. If the table contains no unused columns, the statement returns with no errors.

column

specifies one or more columns to be set as unused or dropped. Use the COLUMN keyword only if you are specifying only one column. If you specify a column list, it cannot contain duplicates.

CASCADE CONSTRAINTS	drops all referential integrity constraints that refer to the primary and unique keys defined on the dropped columns, and drops all multicolumn constraints defined on the dropped columns. If any constraint is referenced by columns from other tables or remaining columns in the target table, then you must specify CASCADE CONSTRAINTS. Otherwise, the statement aborts and an error is returned.
INVALIDATE	**Note:** Currently, Oracle executes this clause regardless of whether you specify the keyword INVALIDATE. Oracle invalidates all dependent objects, such as views, triggers, and stored program units. Object invalidation is a recursive process. Therefore, all directly dependent *and* indirectly dependent objects are invalidated. However, only local dependencies are invalidated, because Oracle manages remote dependencies differently from local dependencies. For more information on dependencies, refer to *Oracle8i Concepts*. An object invalidated by this statement is automatically revalidated when next referenced. You must then correct any errors that exist in that object before referencing it.
CHECKPOINT	specifies that a checkpoint for the drop column operation will be applied after processing *integer* rows; *integer* is optional and must be greater than zero. If *integer* is greater than the number of rows in the table, Oracle applies a checkpoint after all the rows have been processed. If you do not specify *integer*, Oracle sets the default of 512. Checkpointing cuts down the amount of undo logs accumulated during the drop column operation to avoid running out of rollback segment space. However, if this statement is interrupted after a checkpoint has been applied, the table remains in an unusable state. While the table is unusable, the only operations allowed on it are DROP TABLE, TRUNCATE TABLE, and ALTER TABLE DROP COLUMNS CONTINUE (described below). You cannot use this clause with SET UNUSED, because that clause does not remove column data.
DROP COLUMNS CONTINUE	continues the drop column operation from the point at which it was interrupted. Submitting this statement while the table is in a valid state results in an error.

Restrictions on the *drop_column_clause*:

- Each of the parts of this clause can be specified only once in the statement and cannot be mixed with any other ALTER TABLE clauses. For example, the following statements are not allowed:

```
ALTER TABLE t1 DROP COLUMN f1 DROP (f2);

ALTER TABLE t1 DROP COLUMN f1 SET UNUSED (f2);

ALTER TABLE t1 DROP (f1) ADD (f2 NUMBER);

ALTER TABLE t1 SET UNUSED (f3)
   ADD (CONSTRAINT ck1 CHECK (f2 > 0));
```

- You can drop an object type column only as an entity. Dropping an attribute from an object type column is not allowed.
- If you drop a nested table column, its storage table is removed.
- If you drop a LOB column, the LOB data and its corresponding LOB index segment are removed.
- If you drop a BFILE column, only the locators stored in that column are removed, not the files referenced by the locators.
- You can drop a column from an index-organized table only if it is not a primary key column. The primary key constraint of an index-organized table can never be dropped, so you cannot drop a primary key column even if you have specified CASCADE CONSTRAINTS.
- You can export tables with dropped or unused columns. However, you can import a table only if all the columns specified in the export files are present in the table (that is, none of those columns has been dropped or marked unused). Otherwise, Oracle returns an error.
- You cannot drop a column on which a domain index has been built.

You cannot use this clause to drop:

- A pseudocolumn, clustered column, or partitioning column. (You can drop nonpartitioning columns from a partitioned table if all the tablespaces where the partitions were created are online and in read-write mode.)
- A column from a nested table, an object table, or a table owned by SYS

allocate_extent_clause explicitly allocates a new extent for the table, the partition or subpartition, the overflow data segment, the LOB data segment, or the LOB index.

Restriction: You cannot allocate an extent for a composite-partitioned table.

SIZE specifies the size of the extent in bytes. Use K or M to specify the extent size in kilobytes or megabytes. If you omit this parameter, Oracle determines the size based on the values of the STORAGE parameters of the table's overflow data segment or of the LOB index.

DATAFILE — specifies one of the datafiles in the tablespace of the table, overflow data segment, LOB data tablespace, or LOB index to contain the new extent. If you omit this parameter, Oracle chooses the datafile.

INSTANCE — makes the new extent available to the freelist group associated with the specified instance. If the instance number exceeds the maximum number of freelist groups, the former is divided by the latter, and the remainder is used to identify the freelist group to be used. An instance is identified by the value of its initialization parameter INSTANCE_NUMBER. If you omit this parameter, the space is allocated to the table, but is not drawn from any particular freelist group. Rather, the master freelist is used, and space is allocated as needed. For more information, see *Oracle8i Concepts.* **Use this parameter only if you are using Oracle with the Parallel Server option in parallel mode.**

Explicitly allocating an extent with this clause does affect the size for the next extent to be allocated as specified by the NEXT and PCTINCREASE storage parameters.

deallocate_unused_clause — explicitly deallocates unused space at the end of the table, partition or subpartition, overflow data segment, LOB data segment, or LOB index and makes the space available for other segments in the tablespace. You can free only unused space above the high water mark (that is, the point beyond which database blocks have not yet been formatted to receive data).

Oracle credits the amount of the released space to the user quota for the tablespace in which the deallocation occurs.

Oracle deallocates unused space from the end of the object toward the high water mark at the beginning of the object. If an extent is completely contained in the deallocation, then the whole extent is freed for reuse. If an extent is partially contained in the deallocation, then the used part up to the high water mark becomes the extent, and the remaining unused space is freed for reuse.

The exact amount of space freed depends on the values of the INITIAL, MINEXTENTS, and NEXT parameters (as described in "storage_clause" on page 7-575).

KEEP — specifies the number of bytes above the high water mark that the table, overflow data segment, LOB data segment, or LOB index will have after deallocation.

- If you omit KEEP and the high water mark is above the size of INITIAL and MINEXTENTS, then all unused space above the high water mark is freed. When the high water mark is less than the size of INITIAL or MINEXTENTS, then all unused space above MINEXTENTS is freed.
- If you specify KEEP, then the specified amount of space is kept and the remaining space is freed. When the remaining number of extents is less than MINEXTENTS, then MINEXTENTS is adjusted to the new number of extents. If the initial extent becomes smaller than INITIAL, then INITIAL is adjusted to the new size.
- In either case, NEXT is set to the size of the last extent that was deallocated.

CACHE — for data that is accessed frequently, specifies that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This attribute is useful for small lookup tables.

Restriction: You cannot specify CACHE for index-organized tables.

NOCACHE — for data that is not accessed frequently, specifies that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed.

For LOBs, the LOB value is either not brought into the buffer cache or brought into the buffer cache and placed at the least recently used end of the LRU list. (The latter is the default behavior.)

Restriction: You cannot specify NOCACHE for index-organized tables.

MONITORING — specifies that Oracle can collect modification statistics on *table*. These statistics are estimates of the number of rows affected by DML statements over a particular period of time. They are available for use by the optimizer or for analysis by the user.

For more information on using this clause, see *Oracle8i Tuning*.

NOMONITORING — specifies that Oracle will not collect modification statistics on *table*.

Restriction: You cannot specify MONITORING or NOMONITORING for a temporary table.

LOGGING | NOLOGGING — specifies whether subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against a nonpartitioned table, table partition, all partitions of a partitioned table, or all subpartitions of a partition will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file.

When used with the *modify_default_attributes_clause*, this clause affects the logging attribute of a partitioned table.

LOGGING | NOLOGGING also specifies whether ALTER TABLE...MOVE and ALTER TABLE...SPLIT operations will be logged or not logged.

In NOLOGGING mode, data is modified with minimal logging (to mark new extents invalid and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not logged. Therefore, if you cannot afford to lose this table, it is important to take a backup after the NOLOGGING operation.

If the database is run in ARCHIVELOG mode, media recovery from a backup taken before the LOGGING operation will restore the table. However, media recovery from a backup taken before the NOLOGGING operation will not restore the table.

The logging attribute of the base table is independent of that of its indexes.

For more information about the *logging_clause* and parallel DML, see *Oracle8i Parallel Server Concepts and Administration*.

RENAME TO — renames *table* to *new_table_name*.

Note: Using this clause will invalidate any dependent materialized views. For more information on materialized views, see "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 7-300 and *Oracle8i Tuning*.

records_per_block_clause — determines whether Oracle restricts the number of records that can be stored in a block. This clause ensures that any bitmap indexes subsequently created on the table will be as small (compressed) as possible.

Restrictions:

- You cannot specify either MINIMIZE or NOMINIMIZE if a bitmap index has already been defined on *table*. You must first drop the bitmap index.
- You cannot specify this clause for an index-organized table or nested table.

MINIMIZE — instructs Oracle to calculate the largest number of records in any block in the table, and limit future inserts so that no block can contain more than that number of records.

Restriction: You cannot specify MINIMIZE for an empty table.

NOMINIMIZE — disables the MINIMIZE feature. This is the default.

alter_overflow_clause — modifies the definition of an index-organized table. Index-organized tables keep data sorted on the primary key and are therefore best suited for primary-key-based access and manipulation.

Note: When you alter an index-organized table, Oracle evaluates the maximum size of each column to estimate the largest possible row. If an overflow segment is needed but you have not specified OVERFLOW, Oracle raises an error and does not execute the ALTER TABLE statement. This checking function guarantees that subsequent DML operations on the index-organized table will not fail because an overflow segment is lacking.

PCTTHRESHOLD *integer*

specifies the percentage of space reserved in the index block for an index-organized table row. Any portion of the row that exceeds the specified threshold is stored in the overflow area. PCTTHRESHOLD must be a value from 1 to 50.

Restrictions:

- You cannot reduce the value of PCTTHRESHOLD so much that the primary key will not fit.
- You cannot specify PCTTHRESHOLD for individual partitions of an index-organized table.

INCLUDING *column_name*

specifies the column at which to divide an index-organized table row into index and overflow portions. All non-primary-key columns that follow *column_name* are stored in the overflow data segment. The *column_name* is either the name of the last primary key column or any subsequent non-primary-key column.

If you use the *drop_column_clause* to drop (or mark unused) a column defined as an INCLUDING column, the column stored immediately before this column will become the new INCLUDING column.

overflow_clause

specifies the overflow data segment physical storage and logging attributes to be modified for the index-organized table. Parameters specified in this clause are applicable only to the overflow data segment. For more information, see "CREATE TABLE" on page 7-359.

Restriction: You cannot specify OVERFLOW for a partition of a partitioned index-organized table unless the table already has an overflow segment.

add_overflow_clause

adds an overflow data segment to the specified index-organized table.

For a partitioned index-organized table:

- If you do not specify PARTITION, Oracle automatically allocates an overflow segment for each partition. The physical attributes of these segments are inherited from the table level.
- If you wish to specify separate physical attributes for one or more partitions, you must specify such attributes for **every** partition in the table. You do not specify the name of the partitions, but you must specify their attributes in the order in which they were created.

  You can find the order of the partitions by querying the PARTITION_NAME and PARTITION_POSITION columns of the USER_IND_PARTITIONS view.

If you do not specify TABLESPACE for a particular partition, Oracle uses the tablespace specified for the table. If you do not specify TABLESPACE at the table level, Oracle uses the tablespace of the partition's primary key index segment.

partitioning_clauses

The following clauses apply only to partitioned tables. You cannot combine partition operations with other partition operations or with operations on the base table in one ALTER TABLE statement.

Note: If you drop, exchange, truncate, move, modify, or split a partition on a table that is a master table for one or more materialized views, existing bulk load information about the table will be deleted. Therefore, be sure to refresh all dependent materialized views before performing any of these operations.

modify_default_attributes_clause

specifies new default values for the attributes of *table*. Partitions and LOB partitions you create subsequently will inherit these values unless you override them explicitly when creating the partition or LOB partition. Existing partitions and LOB partitions are not affected by this clause.

Only attributes named in the statement are affected, and the default values specified are overridden by any attributes specified at the individual partition level.

FOR PARTITION applies only to composite-partitioned tables. This clause specifies new default values for the attributes of *partition*. Subpartitions and LOB subpartitions of *partition* that you create subsequently will inherit these values, unless you override them explicitly when creating the subpartition or LOB subpartition. Existing subpartitions are not affected by this clause.

Restrictions:

- The PCTTHRESHOLD, COMPRESS, *physical_attributes_clause*, and *overflow_clause* are valid only for partitioned index-organized tables.
- You cannot specify the PCTUSED parameter for the index segment of an index-organized table.
- You can specify COMPRESS only if compression is already specified at the table level.

modify_partition_clause

modifies the real physical attributes of the *partition* table partition. Optionally modifies the storage attributes of one or more LOB items for the partition. You can specify new values for any of the following physical attributes for the partition: the logging attribute; PCTFREE, PCTUSED, INITRANS, or MAXTRANS parameter; or storage parameters.

If *table* is composite-partitioned:

- If you specify the *allocate_extent_clause*, Oracle will allocate an extent for each subpartition of *partition*.
- If you specify *deallocate_unused_clause*, Oracle will deallocate unused storage from each subpartition of *partition*.
- Any other attributes changed in this clause will be changed in subpartitions of *partition* as well, overriding existing values. To avoid changing the attributes of existing subpartitions, use the FOR PARTITION clause of the *modify_default_attributes_clause*.

Restriction: If *table* is hash partitioned, you can specify only the *allocate_extent* and *deallocate_unused* clauses. All other attributes of the partition are inherited from the table-level defaults except TABLESPACE, which stays the same as it was at create time.

add_subpartition_clause adds a hash subpartition to *partition*. Oracle populates the new subpartition with rows rehashed from the other subpartition(s) of *partition* as determined by the hash function.

Oracle marks UNUSABLE, and you must rebuild, the local index subpartitions corresponding to the added and to the rehashed subpartitions.

If you do not specify *subpartition*, Oracle assigns a name in the form SYS_SUBP*nnnn*

If you do not specify TABLESPACE, the new subpartition will reside in the default tablespace of *partition*.

COALESCE SUBPARTITION specifies that Oracle should select a hash subpartition, distribute its contents into one or more remaining subpartitions (determined by the hash function), and then drop the selected subpartition.

Local index subpartitions corresponding to the selected subpartition are also dropped. Oracle marks UNUSABLE, and you must rebuild, the index subpartitions corresponding to one or more absorbing subpartitions.

UNUSABLE LOCAL INDEXES clause The next two clauses modify the attributes of local **index partitions** corresponding to *partition*.

UNUSABLE LOCAL INDEXES marks UNUSABLE all the local index partitions associated with *partition*.

REBUILD UNUSABLE LOCAL INDEXES rebuilds the unusable local index partitions associated with *partition*.

Restrictions:

- You cannot specify this clause with any other clauses of the *modify_partition_clause*.
- You cannot specify this clause for partitions that are subpartitioned.

modify_subpartition_clause lets you allocate or deallocate storage for an individual subpartition of *table*.

Restriction: The only *modify_LOB_storage_parameters* you can specify for subpartition are the *allocate_extent_clause* and *deallocate_unused_clause*.

UNUSABLE LOCAL INDEXES marks UNUSABLE all the local index subpartitions associated with *subpartition*.

REBUILD UNUSABLE LOCAL INDEXES rebuilds the unusable local index subpartitions associated with sub*partition*.

rename_partition/ subpartition_ clause renames a table partition or subpartition *current_name* to *new_name*. For both partitions and subpartitions, *new_name* must be different from all existing partitions and subpartitions of the same table.

move_partition_ clause moves table partition *partition* to another segment. You can move partition data to another tablespace, recluster data to reduce fragmentation, or change create-time physical attributes.

If the table contains LOB columns, you can use the *LOB_storage_clause* to move the LOB data and LOB index segments associated with this partition. Only the LOBs named are affected. If you do not specify the *LOB_storage_clause* for a particular LOB column, its LOB data and LOB index segments are not moved.

If *partition* is not empty, MOVE PARTITION marks UNUSABLE all corresponding local index partitions and all global nonpartitioned indexes, and all the partitions of global partitioned indexes.

When you move a LOB data segment, Oracle drops the old data segment and corresponding index segment and creates new segments even if you do not specify a new tablespace.

The move operation obtains its parallel attribute from the *parallel_clause*, if specified. If not specified, the default parallel attributes of the table, if any, are used. If neither is specified, Oracle performs the move without using parallelism.

The *parallel_clause* on MOVE PARTITION does not change the default parallel attributes of *table*.

Note: For index-organized tables, Oracle uses the address of the primary key, as well as its value, to construct logical rowids. The logical rowids are stored in the secondary index of the table. If you move a partition of an index-organized table, the address portion of the rowids will change, which can hamper performance. To ensure optimal performance, rebuild the secondary index(es) on the moved partition to update the rowids. For more information on logical rowids, see *Oracle8i Concepts*.

Restrictions:

- If *partition* is a hash partition, the only attribute you can specify in this clause is TABLESPACE.
- You cannot move a partition of a composite-partitioned table. You must move each subpartition separately with the *move_subpartition_clause*.
- You cannot specify this clause for a partition containing subpartitions. However, you can move subpartitions using the *move_subpartition_clause*.

move_ subpartition_ clause

moves the table subpartition *subpartition* to another segment. If you do not specify TABLESPACE, the subpartition will remain in the same tablespace.

Unless the subpartition is empty, Oracle marks UNUSABLE all local index subpartitions corresponding to the subpartition being moved, as well as global nonpartitioned indexes and partitions of global indexes.

If the table contains LOB columns, you can use the *LOB_storage_clause* to move the LOB data and LOB index segments associated with this subpartition. Only the LOBs named are affected. If you do not specify the *LOB_storage_clause* for a particular LOB column, its LOB data and LOB index segments are not moved.

When you move a LOB data segment, Oracle drops the old data segment and corresponding index segment and creates new segments even if you do not specify a new tablespace.

add_range_ partition_clause

adds a new range partition *partition* to the "high" end of a partitioned table (after the last existing partition). You can specify any create-time physical attributes for the new partition. If the table contains LOB columns, you can also specify partition-level attributes for one or more LOB items.

You can specify up to 64K-1 partitions. For a discussion of factors that might impose practical limits less than this number, refer to *Oracle8i Administrator's Guide.*

Restrictions:

- If the first element of the partition bound of the high partition is MAXVALUE, you cannot add a partition to the table. Instead, use the *split_partition_clause* to add a partition at the beginning or the middle of the table.
- The *compression_clause*, *physical_attributes_clause*, and OVERFLOW are valid only for a partitioned index-organized table.
- You cannot specify the PCTUSED parameter for the index segment of an index-organized table.
- You can specify OVERFLOW only if the partitioned table already has an overflow segment.
- You can specify compression only if compression is enabled at the table level.

VALUES LESS THAN (*value_list*)

specifies the upper bound for the new partition. The *value_list* is a comma-separated, ordered list of literal values corresponding to *column_list*. The *value_list* must collate greater than the partition bound for the highest existing partition in the table.

partition_level_ subpartitioning

is permitted only for a composite-partitioned table. This clause lets you specify particular hash subpartitions for *partition*. You specify composite partitioning in one of two ways:

- You can specify individual subpartitions by name, and optionally the tablespace where each should be stored, or
- You can specify the number of subpartitions (and optionally one or more tablespaces where they are to be stored). In this case, Oracle assigns partition names of the form SYS_SUBP*nnn*. The number of tablespaces does not have to equal the number of subpartitions. If the number of subpartitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.

The subpartitions inherit all their attributes from any attributes specified for *new_partition*, except for TABLESPACE, which you can specify at the subpartition level. Any attributes not specified at the subpartition or partition level are inherited from table-level defaults.

This clause overrides any subpartitioning specified at the table level.

If you do not specify this clause but you specified default subpartitioning at the table level, *new_partition_name* will inherit the table-level default subpartitioning (see "CREATE TABLE" on page 7-359).

add_hash_partition_clause adds a new hash partition to the "high" end of a partitioned table. Oracle will populate the new partition with rows rehashed from other partitions of *table* as determined by the hash function.

You can specify a name for the partition, and optionally a tablespace where it should be stored. If you do not specify *new_partition_name*, Oracle assigns a partition name of the form SYS_P*nnn*. If you do not specify TABLESPACE, the new partition is stored in the table's default tablespace. Other attributes are always inherited from table-level defaults.

For more information on hash partitioning, see "CREATE TABLE" on page 7-359 and *Oracle8i Concepts*.

parallel_clause lets you specify whether to parallelize the creation of the new partition.

coalesce_partition_clause applies only to hash-partitioned tables. This clause specifies that Oracle should select a hash partition, distribute its contents into one or more remaining partitions (determined by the hash function), and then drop the selected partition. Local index partitions corresponding to the selected partition are also dropped. Oracle marks UNUSABLE, and you must rebuild, the local index partitions corresponding to one or more absorbing partitions.

drop_partition_clause applies only to tables partitioned using the range or composite method. This clause removes partition *partition*, and the data in that partition, from a partitioned table. If you want to drop a partition but keep its data in the table, you must merge the partition into one of the adjacent partitions. See the *merge_partitions_clause* of this statement.

If the table has LOB columns, the LOB data and LOB index partitions (and their subpartitions, if any) corresponding to *partition* are also dropped.

- Oracle drops local index partitions and subpartitions corresponding to *partition*, even if they are marked UNUSABLE.
- Oracle marks UNUSABLE all global nonpartitioned indexes defined on the table and all partitions of global partitioned indexes, unless the partition being dropped or all of its subpartitions are empty.
- If you drop a partition and later insert a row that would have belonged to the dropped partition, Oracle stores the row in the next higher partition. However, if that partition is the highest partition, the insert will fail because the range of values represented by the dropped partition is no longer valid for the table.

Restriction: If *table* contains only one partition, you cannot drop the partition. You must drop the table.

truncate_partition_clause

PARTITION removes all rows from *partition* or, if the table is composite-partitioned, all rows from *partition*'s subpartitions. SUBPARTITION removes all rows from *subpartition*.

truncate_subpartition_clause

If the table contains any LOB columns, the LOB data and LOB index segments for this partition are also truncated. If the table is composite-partitioned, the LOB data and LOB index segments for this partition's subpartitions are truncated.

If the partition or subpartition to be truncated contains data, you must first disable any referential integrity constraints on the table. Alternatively, you can delete the rows and then truncate the partition.

For each partition or subpartition truncated, Oracle also truncates corresponding local index partitions and subpartitions. If those index partitions or subpartitions are marked UNUSABLE, Oracle truncates them and resets the UNUSABLE marker to VALID. In addition, if the truncated partition or subpartition, or any of the subpartitions of the truncated partition are not empty, Oracle marks as UNUSABLE **all** global nonpartitioned indexes and partitions of global indexes defined on the table.

DROP STORAGE — deallocates space from the deleted rows and makes it available for use by other schema objects in the tablespace.

REUSE STORAGE — keeps space from the deleted rows allocated to the partition or subpartition. The space is subsequently available only for inserts and updates to the same partition or subpartition.

split_partition_clause

from an original partition *partition_name_old*, creates two new partitions, each with a new segment and new physical attributes, and new initial extents. The segment associated with *partition_name_old* is discarded.

Restriction: You cannot specify this clause for a hash-partitioned table.

AT (*value_list*) — specifies the new noninclusive upper bound for *split_partition_1*. The *value_list* must compare less than the original partition bound for *partition_name_old* and greater than the partition bound for the next lowest partition (if there is one).

INTO	describes the two partitions resulting from the split.
partition_description, partition_description	specifies optional names and physical attributes of the two partitions resulting from the split. If you do not specify new partition names, Oracle assigns names of the form SYS_P*n*. Any attributes you do not specify are inherited from *partition_name_old*. **Restriction**: ■ You can specify the *compression_clause*, *physical_attributes_clause*, and OVERFLOW only for a partitioned index-organized table. ■ You cannot specify the PCTUSED parameter for the index segment of an index-organized table.
parallel_clause	parallelizes the split operation, but does not change the default parallel attributes of the table.

If you specify subpartitioning for the new partitions, you can specify only TABLESPACE for the subpartitions. All other attributes will be inherited from the containing new partition.

If *partition_name_old* is subpartitioned, and you do not specify any subpartitioning for the new partitions, the new partitions will inherit the number and tablespaces of the subpartitions in *partition_name_old*.

Oracle also splits corresponding local index partitions, even if they are marked UNUSABLE. The resulting local index partitions inherit all their partition-level default attributes from the local index partition being split.

If *partition_name_old* was not empty, Oracle marks UNUSABLE **all** global nonpartitioned indexes and all partitions of global indexes on the table. (This action on global indexes does not apply to index-organized tables.) In addition, if any partitions or subpartitions resulting from the split are not empty, Oracle marks as UNUSABLE all corresponding local index partitions and subpartitions.

If *table* contains LOB columns, you can use the *LOB_storage_clause* to specify separate LOB storage attributes for the LOB data segments resulting from the split. Oracle drops the LOB data and LOB index segments of *partition_name_old* and creates new segments for each LOB column, for each partition, even if you do not specify a new tablespace.

merge_partitions_clause merges the contents of two adjacent partitions of *table* into one new partition, and then drops the original two partitions.

The new partition inherits the partition-bound of the higher of the two original partitions.

Any attributes not specified in the *segment_attributes_clause* are inherited from table-level defaults.

If you do not specify *new_partition_name*, Oracle assigns a name of the form SYS_P*nnn*. If the new partition has subpartitions, Oracle assigns subpartition names of the form SYS_SUBP*nnnn*.

If either or both of the original partitions was not empty, Oracle marks UNUSABLE **all** global nonpartitioned global indexes and all partitions of global indexes on the table. In addition, if the partition or any of its subpartitions resulting from the merge is not empty, Oracle marks UNUSABLE all corresponding local index partitions and subpartitions.

Restriction: You cannot specify this clause for an index-organized table or for a table partitioned using the hash method.

partition_level_partitioning — specifies hash subpartitioning attributes for the new partition. Any attributes not specified in this clause are inherited from table-level defaults.

If you do not specify this clause, the new merged partition inherits subpartitioning attributes from table-level defaults.

parallel_clause — specifies that the merging operation is to be parallelized.

exchange_partition_clause

exchange_subpartition_clause

converts a partition (or subpartition) into a nonpartitioned table, and a nonpartitioned table into a partition (or subpartition) of a partitioned table by exchanging their data (and index) segments. The default behavior is EXCLUDING INDEXES WITH VALIDATION. You must have ALTER TABLE privileges on **both** tables to perform this operation.

This clause facilitates high-speed data loading when used with transportable tablespaces. For information on this topic, see *Oracle8i Administrator's Guide.*

If *table* contains LOB columns, for each LOB column Oracle exchanges LOB data and LOB index partition or subpartition segments with corresponding LOB data and LOB index segments of *table*.

All statistics of the table and partition are exchanged, including table, column, index statistics, and histograms. The aggregate statistics of the partitioned table are recalculated.

The logging attribute of the table and partition is also exchanged.

WITH TABLE *table* — specifies the table with which the partition will be exchanged.

INCLUDING INDEXES — specifies that the local index partitions or subpartitions should be exchanged with the corresponding regular indexes.

EXCLUDING INDEXES — specifies that all the local index partitions or subpartitions corresponding to the partition and all the regular indexes on the exchanged table are marked UNUSABLE.

WITH VALIDATION — specifies that if any rows in the exchanged table do not map into partitions or subpartitions being exchanged, Oracle should return an error.

WITHOUT VALIDATION — specifies that the proper mapping of rows in the exchanged table is not checked.

EXCEPTIONS INTO

This clause applies only to loading a nonpartitioned table into a partitioned table. It lets you specify a table into which Oracle places the rowids of all rows violating the partitioned table's UNIQUE constraint. The script used to create such a table is UTLEXCPT1.SQL.

Note: You can use the UTLEXCPT1.SQL script with index-organized tables. You could not use earlier versions of the script for this purpose. See *Oracle8i Migration* for compatibility information.

Restrictions:

- This clause is not valid with subpartitions.
- The partitioned table must have been defined with a UNIQUE constraint, and that constraint must be in DISABLE VALIDATE state.

If these conditions are not true, Oracle ignores this clause.

For more information on constraint checking, see the "constraint_ clause" on page 7-217.

Restrictions: For partitioned index-organized tables, the following restrictions apply:

- The source and target table / partition must have their primary key set on the same columns, in the same order.
- If compression is enabled, it must be enabled for both the source and the target, and with the same prefix length.
- An index-organized table partition cannot be exchanged with a regular table or vice versa.
- Both the source and target must have overflow segments, or neither can have overflow segments.

row_movement_ clause

determines whether a row can be moved to a different partition or subpartition because of a change to one or more of its key values.

Restriction: You can specify this clause only for a partitioned table.

ENABLE

allows Oracle to move a row to a different partition or subpartition as the result of an update to the partitioning or subpartitioning key.

Restriction: You cannot specify this clause if a domain index has been built on any column of the table.

WARNING: Moving a row in the course of an UPDATE operation changes that row's ROWID.

DISABLE

returns an error if an update to a partitioning or subpartitioning key would result in a row moving to a different partition or subpartition. This is the default.

parallel_clause	changes the default degree of parallelism for queries and DML on the table. For additional information, see the Notes to the *parallel_clause* of "CREATE TABLE" on page 7-359. NOPARALLEL — specifies serial execution. This is the default. PARALLEL — causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter. PARALLEL *integer* — specifies the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution processes. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*. **Restriction:** If *table* contains any columns of LOB or user-defined object type, subsequent INSERT, UPDATE, and DELETE operations on *table* are executed serially without notification. Subsequent queries, however, will be executed in parallel. **Note:** If you specify the *parallel_clause* in conjunction with the *move_table_clause*, the parallelism applies only to the move, not to subsequent DML and query operations on the table.
enable_disable_clause	lets you specify whether Oracle should apply an integrity constraint. For a complete description of this clause, including notes and restrictions that relate to this statement, see the *enable_disable_clause* of "CREATE TABLE" on page 7-359.
ENABLE TABLE LOCK	enables DML and DDL locks on a table in a parallel server environment. For more information, see *Oracle8i Parallel Server Concepts and Administration*. **Note:** DML table locks are not acquired on temporary tables.
DISABLE TABLE LOCK	disables DML and DDL locks on a table to improve performance in a parallel server environment. For more information, see *Oracle8i Parallel Server Concepts and Administration*.
ENABLE ALL TRIGGERS	enables all triggers associated with the table. Oracle fires the triggers whenever their triggering condition is satisfied. See "CREATE TRIGGER" on page 7-401. To enable a single trigger, use the *enable_clause* of ALTER TRIGGER. See "ALTER TRIGGER" on page 7-171.
DISABLE ALL TRIGGERS	disables all triggers associated with the table. Oracle will not fire a disabled trigger even if the triggering condition is satisfied.

Examples

Nested Table Example The following statement modifies the storage characteristics of a nested table column PROJECTS in table EMP so that when queried it returns actual values instead of locators:

```
ALTER TABLE emp MODIFY NESTED TABLE projects RETURN AS VALUE;
```

PARALLEL Example The following statement specifies parallel processing for queries to the EMP table:

```
ALTER TABLE emp
   PARALLEL;
```

ENABLE VALIDATE Example The following statement places in ENABLE VALIDATE state an integrity constraint named FK_DEPTNO in the EMP table:

```
ALTER TABLE emp
   ENABLE VALIDATE CONSTRAINT fk_deptno
   EXCEPTIONS INTO except_table;
```

Each row of the EMP table must satisfy the constraint for Oracle to enable the constraint. If any row violates the constraint, the constraint remains disabled. Oracle lists any exceptions in the table EXCEPT_TABLE. You can also identify the exceptions in the EMP table with the following statement:

```
SELECT emp.*
   FROM emp e, except_table ex
   WHERE e.row_id = ex.row_id
      AND ex.table_name = 'EMP'
      AND ex.constraint = 'FK_DEPTNO';
```

ENABLE NOVALIDATE Example The following statement tries to place in ENABLE NOVALIDATE state two constraints on the EMP table:

```
ALTER TABLE emp
   ENABLE NOVALIDATE UNIQUE (ename)
   ENABLE NOVALIDATE CONSTRAINT nn_ename;
```

This statement has two ENABLE clauses:

- The first places a unique constraint on the ENAME column in ENABLE NOVALIDATE state.
- The second places the constraint named NN_ENAME in ENABLE NOVALIDATE state.

In this case, Oracle enables the constraints only if both are satisfied by each row in the table. If any row violates either constraint, Oracle returns an error and both constraints remain disabled.

DISABLE Example Consider a referential integrity constraint involving a foreign key on the combination of the AREACO and PHONENO columns of the PHONE_CALLS table. The foreign key references a unique key on the combination of the AREACO and PHONENO columns of the CUSTOMERS table. The following statement disables the unique key on the combination of the AREACO and PHONENO columns of the CUSTOMERS table:

```
ALTER TABLE customers
   DISABLE UNIQUE (areaco, phoneno) CASCADE;
```

The unique key in the CUSTOMERS table is referenced by the foreign key in the PHONE_CALLS table, so you must use the CASCADE clause to disable the unique key. This clause disables the foreign key as well.

CHECK Example The following statement defines and disables a CHECK constraint on the EMP table:

```
ALTER TABLE emp
   ADD (CONSTRAINT check_comp CHECK (sal + comm <= 5000) )
   DISABLE CONSTRAINT check_comp;
```

The constraint CHECK_COMP ensures that no employee's total compensation exceeds $5000. The constraint is disabled, so you can increase an employee's compensation above this limit.

Triggers Example The following statement enables all triggers associated with the EMP table:

```
ALTER TABLE emp
   ENABLE ALL TRIGGERS;
```

DEALLOCATE UNUSED Example The following statement frees all unused space for reuse in table EMP, where the high water mark is above MINEXTENTS:

```
ALTER TABLE emp
    DEALLOCATE UNUSED;
```

DROP COLUMN Example This statement illustrates the *drop_column_clause* with CASCADE CONSTRAINTS. Assume table T1 is created as follows:

```
CREATE TABLE t1 (
```

```
   pk NUMBER PRIMARY KEY,
   fk NUMBER,
   c1 NUMBER,
   c2 NUMBER,
   CONSTRAINT ri FOREIGN KEY (fk) REFERENCES t1,
   CONSTRAINT ck1 CHECK (pk > 0 and c1 > 0),
   CONSTRAINT ck2 CHECK (c2 > 0)
);
```

An error will be returned for the following statements:

```
ALTER TABLE t1 DROP (pk); -- pk is a parent key
ALTER TABLE t1 DROP (c1);  -- c1 is referenced by multicolumn
                           constraint ck1
```

Submitting the following statement drops column PK, the primary key constraint, the foreign key constraint, RI, and the check constraint, CK1:

```
ALTER TABLE t1 DROP (pk) CASCADE CONSTRAINTS;
```

If all columns referenced by the constraints defined on the dropped columns are also dropped, then CASCADE CONSTRAINTS is not required. For example, assuming that no other referential constraints from other tables refer to column PK, then it is valid to submit the following statement without the CASCADE CONSTRAINTS clause:

```
ALTER TABLE t1 DROP (pk, fk, c1);
```

Index-Organized Table Examples This statement modifies the INITRANS parameter for the index segment of index-organized table DOCINDEX:

```
ALTER TABLE docindex INITRANS 4;
```

The following statement adds an overflow data segment to index-organized table DOCINDEX:

```
ALTER TABLE docindex ADD OVERFLOW;
```

This statement modifies the INITRANS parameter for the overflow data segment of index-organized table DOCINDEX:

```
ALTER TABLE docindex OVERFLOW INITRANS 4;
```

ADD PARTITION Example The following statement adds a partition P3 and specifies storage characteristics for three of the table's LOB columns (B, C, and D):

```
ALTER TABLE pt ADD PARTITION p3 VALUES LESS THAN (30)
```

```
   LOB (b, d) STORE AS (TABLESPACE tsz)
   LOB (c) STORE AS mylobseg;
```

The LOB data and LOB index segments for columns B and D in partition P3 will reside in tablespace TSZ. The remaining attributes for these LOB columns will be inherited first from the table-level defaults, and then from the tablespace defaults.

The LOB data segments for column C will reside in the MYLOBSEG segment, and will inherit all other attributes from the table-level defaults and then from the tablespace defaults.

SPLIT PARTITION Example The following statement splits partition P3 into partitions P3_1 and P3_2:

```
ALTER TABLE pt SPLIT PARTITION p3 AT VALUES LESS THAN (25)
   INTO (PARTITION p3_1 TABLESPACE ts4
            LOB (b,d) STORE AS (TABLESPACE tsz),
         PARTITION p3_2 (TABLESPACE ts5)
            LOB (c) STORE AS (TABLESPACE ts5);
```

In partition P3_1, Oracle creates the LOB segments for columns B and D in tablespace TSZ. In partition P3_2, Oracle creates the LOB segments for column C in tablespace TS5. The LOB segments for columns B and D in partition P3_2 and those for column C in partition P3_1 remain in original tablespace for the original partition P3. However, Oracle creates new segments for all the LOB data and LOB index segments, even though they are not moved to a new tablespace.

User-Defined Object Identifier Example The following statements create an object type, a corresponding object table with a primary-key-based object identifier, and a table having a user-defined REF column:

```
CREATE TYPE emp_t AS OBJECT (empno NUMBER, address CHAR(30));

CREATE TABLE emp OF emp_t (
   empno PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;

CREATE TABLE dept (dno NUMBER, mgr_ref REF emp_t SCOPE is emp);
```

The next statements add a constraint and a user-defined REF column, both of which reference table EMP:

```
ALTER TABLE dept ADD CONSTRAINT mgr_cons FOREIGN_KEY (mgr_ref)
   REFERENCES emp;
ALTER TABLE dept ADD sr_mgr REF emp_t REFERENCES emp;
```

Add Column Example The following statement adds a column named THRIFTPLAN of datatype NUMBER with a maximum of seven digits and two decimal places and a column named LOANCODE of datatype CHAR with a size of one and a NOT NULL integrity constraint:

```
ALTER TABLE emp
   ADD (thriftplan NUMBER(7,2),
        loancode CHAR(1) NOT NULL);
```

Modify Column Examples The following statement increases the size of the THRIFTPLAN column to nine digits:

```
ALTER TABLE emp
   MODIFY (thriftplan NUMBER(9,2));
```

Because the MODIFY clause contains only one column definition, the parentheses around the definition are optional.

The following statement changes the values of the PCTFREE and PCTUSED parameters for the EMP table to 30 and 60, respectively:

```
ALTER TABLE emp
    PCTFREE 30
    PCTUSED 60;
```

ALLOCATE EXTENT Example The following statement allocates an extent of 5 kilobytes for the EMP table and makes it available to instance 4:

```
ALTER TABLE emp
  ALLOCATE EXTENT (SIZE 5K INSTANCE 4);
```

Because this statement omits the DATAFILE parameter, Oracle allocates the extent in one of the datafiles belonging to the tablespace containing the table.

DEFAULT Examples This statement modifies the BAL column of the ACCOUNTS table so that it has a default value of 0:

```
ALTER TABLE accounts
  MODIFY (bal DEFAULT 0);
```

If you subsequently add a new row to the ACCOUNTS table and do not specify a value for the BAL column, the value of the BAL column is automatically 0:

```
INSERT INTO accounts(accno, accname)
  VALUES (accseq.nextval, 'LEWIS');
```

```
SELECT *
    FROM accounts
    WHERE accname = 'LEWIS';

ACCNO  ACCNAME BAL
------ ------- ---
815234  LEWIS     0
```

To discontinue previously specified default values, so that they are no longer automatically inserted into newly added rows, replace the values with nulls, as shown in this statement:

```
ALTER TABLE accounts
   MODIFY (bal DEFAULT NULL);
```

The MODIFY clause need only specify the column name and the modified part of the definition, rather than the entire column definition. This statement has no effect on any existing values in existing rows.

Drop Constraint Examples The following statement drops the primary key of the DEPT table:

```
ALTER TABLE dept
    DROP PRIMARY KEY CASCADE;
```

If you know that the name of the PRIMARY KEY constraint is PK_DEPT, you could also drop it with the following statement:

```
ALTER TABLE dept
    DROP CONSTRAINT pk_dept CASCADE;
```

The CASCADE clause drops any foreign keys that reference the primary key.

The following statement drops the unique key on the DNAME column of the DEPT table:

```
ALTER TABLE dept
    DROP UNIQUE (dname);
```

The DROP clause in this statement omits the CASCADE clause. Because of this omission, Oracle does not drop the unique key if any foreign key references it.

LOB Examples The following statement adds CLOB column RESUME to the EMPLOYEE table and specifies LOB storage characteristics for the new column:

```
ALTER TABLE employee ADD (resume CLOB)
```

```
  LOB (resume) STORE AS resume_seg (TABLESPACE resume_ts);
```

To modify the LOB column RESUME to use caching, enter the following statement:

```
ALTER TABLE employee MODIFY LOB (resume) (CACHE);
```

Nested Table Examples The following statement adds the nested table column SKILLS to the EMPLOYEE table:

```
ALTER TABLE employee ADD (skills skill_table_type)
    NESTED TABLE skills STORE AS nested_skill_table;
```

You can also modify a nested table's storage characteristics. Use the name of the storage table specified in the *nested_table_storage_clause* to make the modification. You *cannot* query or perform DML statements on the storage table. Use the storage table only to modify the nested table column storage characteristics.

The following statement creates table VETSERVICE with nested table column CLIENT and storage table CLIENT_TAB. Nested table VETSERVICE is modified to specify constraints:

```
CREATE TYPE pet_table AS OBJECT
   (pet_name VARCHAR2(10), pet_dob DATE);

CREATE TABLE vetservice (vet_name VARCHAR2(30),
                         client   pet_table)
  NESTED TABLE client STORE AS client_tab;

ALTER TABLE client_tab ADD UNIQUE (ssn);
```

The following statement adds a UNIQUE constraint to nested table NESTED_SKILL_TABLE:

```
ALTER TABLE nested_skill_table ADD UNIQUE (a);
```

The following statement alters the storage table for a nested table of REF values to specify that the REF is scoped:

```
CREATE TYPE emp_t AS OBJECT (eno number, ename char(31));
CREATE TYPE emps_t AS TABLE OF REF emp_t;
CREATE TABLE emptab OF emp_t;
CREATE TABLE dept (dno NUMBER, employees emps_t)
   NESTED TABLE employees STORE AS deptemps;
ALTER TABLE deptemps ADD (SCOPE FOR (column_value) IS emptab);
```

Similarly, to specify storing the REF with rowid:

```
ALTER TABLE deptemps ADD (REF(column_value) WITH ROWID);
```

In order to execute these ALTER TABLE statements successfully, the storage table DEPTEMPS must be empty. Also, because the nested table is defined as a table of scalars (REFs), Oracle implicitly provides the column name COLUMN_VALUE for the storage table.

For more information about nested table storage see "CREATE TABLE" on page 7-359. For more information about nested tables, see *Oracle8i Application Developer's Guide - Fundamentals*.

REF Examples In the following statement an object type DEPT_T has been previously defined. Now, create table EMP as follows:

```
CREATE TABLE emp
   (name VARCHAR(100),
    salary NUMBER,
    dept REF dept_t);
```

An object table DEPARTMENTS is created as:

```
CREATE TABLE departments OF dept_t;
```

The DEPT column can store references to objects of DEPT_T stored in any table. If you would like to restrict the references to point only to objects stored in the DEPARTMENTS table, you could do so by adding a scope constraint on the DEPT column as follows:

```
ALTER TABLE emp
    ADD (SCOPE FOR (dept) IS departments);
```

The above ALTER TABLE statement will succeed *only if* the EMP table is empty.

If you want the REF values in the DEPT column of EMP to also store the rowids, issue the following statement:

```
ALTER TABLE emp
   ADD (REF(dept) WITH ROWID);
```

Add Partition Example The following statement adds partition JAN99 to tablespace TSX:

```
ALTER TABLE sales
  ADD PARTITION jan99 VALUES LESS THAN( '970201' )
  TABLESPACE tsx;
```

Drop Partition Example The following statement drops partition DEC98:

```
ALTER TABLE sales DROP PARTITION dec98;
```

Exchange Partition Example The following statement converts partition FEB97 to table SALES_FEB97 without exchanging local index partitions with corresponding indexes on SALES_FEB97 and without verifying that data in SALES_FEB97 falls within the bounds of partition FEB97:

```
ALTER TABLE sales
   EXCHANGE PARTITION feb97 WITH TABLE sales_feb97
   WITHOUT VALIDATION;
```

Modify Partition Examples The following statement marks all the local index partitions corresponding to the NOV96 partition of the SALES table UNUSABLE:

```
ALTER TABLE sales MODIFY PARTITION nov96
   UNUSABLE LOCAL INDEXES;
```

The following statement rebuilds all the local index partitions that were marked UNUSABLE:

```
ALTER TABLE sales MODIFY PARTITION jan97
   REBUILD UNUSABLE LOCAL INDEXES;
```

The following statement changes MAXEXTENTS and logging attribute for partition BRANCH_NY:

```
ALTER TABLE branch MODIFY PARTITION branch_ny
   STORAGE (MAXEXTENTS 75) LOGGING;
```

Move Partition Example The following statement moves partition DEPOT2 to tablespace TS094:

```
ALTER TABLE parts
  MOVE PARTITION depot2 TABLESPACE ts094 NOLOGGING;
```

Rename Partition Examples The following statement renames a table:

```
ALTER TABLE emp RENAME TO employee;
```

In the following statement, partition EMP3 is renamed:

```
ALTER TABLE employee RENAME PARTITION emp3 TO employee3;
```

Split Partition Example The following statement splits the old partition DEPOT4, creating two new partitions, naming one DEPOT9 and reusing the name of the old partition for the other:

```
ALTER TABLE parts
   SPLIT PARTITION depot4 AT ( '40-001' )
   INTO ( PARTITION depot4 TABLESPACE ts009 STORAGE (MINEXTENTS 2),
          PARTITION depot9 TABLESPACE ts010 )
   PARALLEL (10);
```

Truncate Partition Example The following statement deletes all the data in the SYS_P017 partition and deallocates the freed space:

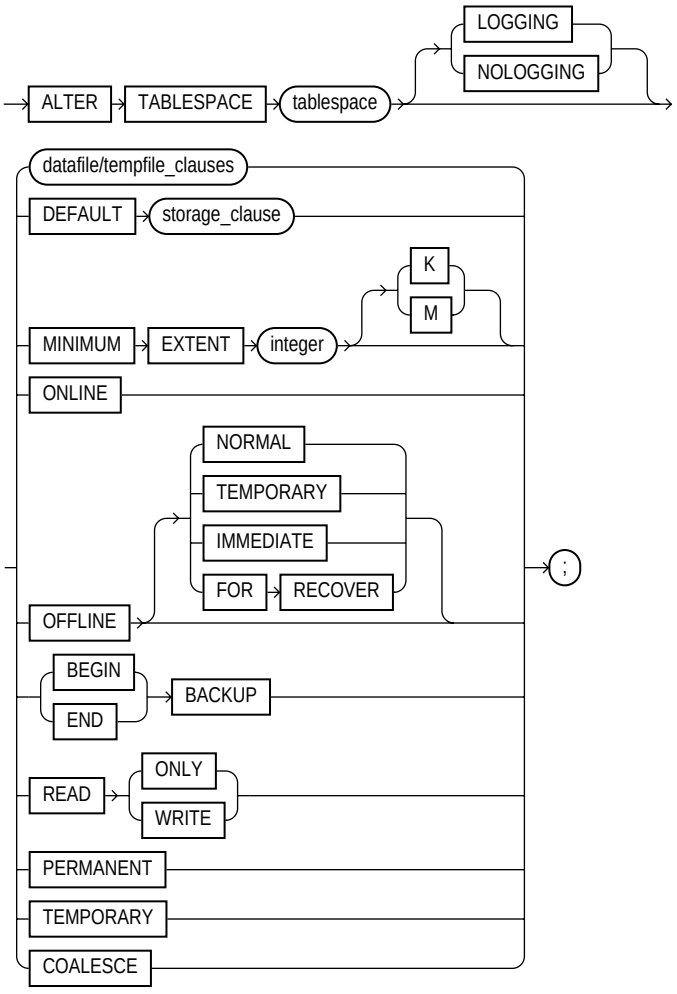
```
ALTER TABLE deliveries
   TRUNCATE PARTITION sys_p017 DROP STORAGE;
```

Additional Examples For examples of defining integrity constraints with the ALTER TABLE statement, see the "constraint_clause" on page 7-217.

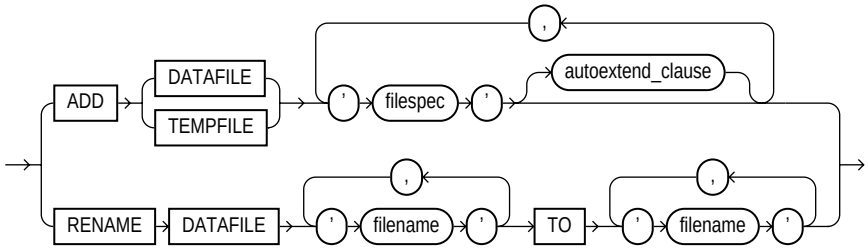
For examples of changing the value of a table's storage parameters, see the "storage_clause" on page 7-575.

ALTER TABLESPACE

Syntax

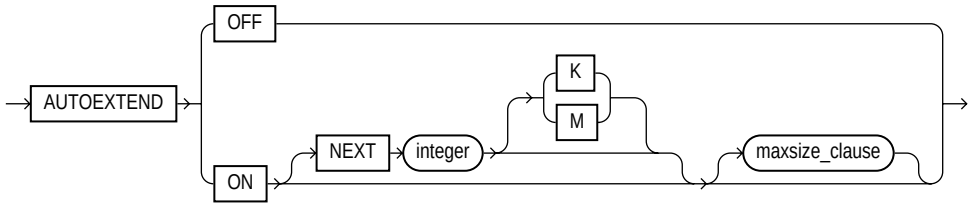


datafile/tempfile_clauses::=

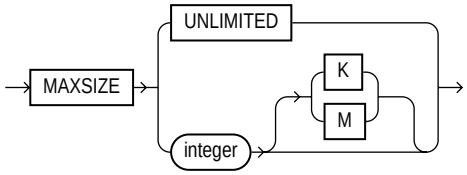


filespec: See "filespec" on page 7-490.

autoextend_clause::=



maxsize_clause::=



storage_clause: See "storage_clause" on page 7-575.

Purpose

To alter an existing tablespace or one or more of its datafiles or tempfiles.

For information on creating a tablespace, see "CREATE TABLESPACE" on page 7-394.

Prerequisites

If you have ALTER TABLESPACE system privilege, you can perform any of this statement's operations. If you have MANAGE TABLESPACE system privilege, you can only perform the following operations:

- take the tablespace online or offline
- begin or end a backup
- make the tablespace read-only or read-write

Before you can make a tablespace read-only, the following conditions must be met:

- The tablespace must be online.
- The tablespace must not contain any active rollback segments. For this reason, the SYSTEM tablespace can never be made read-only, because it contains the SYSTEM rollback segment. Additionally, because the rollback segments of a read-only tablespace are not accessible, Oracle recommends that you drop the rollback segments before you make a tablespace read-only.
- The tablespace must not be involved in an open backup, because the end of a backup updates the header file of all datafiles in the tablespace.

Performing this function in restricted mode may help you meet these restrictions, because only users with RESTRICTED SESSION system privilege can be logged on.

Keywords and Parameters

tablespace	is the name of the tablespace to be altered.
LOGGING | NOLOGGING	specifies the default logging attribute of all tables, indexes, and partitions within the tablespace. The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels. When an existing tablespace logging attribute is changed by an ALTER TABLESPACE statement, all tables, indexes, and partitions created *after* the statement will have the new default logging attribute (which you can still subsequently override). The logging attributes of existing objects are not changed. Only the following operations support NOLOGGING mode: ■ DML: direct-load INSERT (serial or parallel); Direct Loader (SQL*Loader) ■ DDL: CREATE TABLE... AS SELECT, CREATE INDEX, ALTER INDEX... REBUILD, ALTER INDEX... REBUILD PARTITION, ALTER INDEX... SPLIT PARTITION, ALTER TABLE... SPLIT PARTITION, ALTER TABLE... MOVE PARTITION.

In NOLOGGING mode, data is modified with minimal logging (to mark new extents invalid and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not logged. Therefore, if you cannot afford to lose the object, it is important to take a backup after the NOLOGGING operation.

datafile/tempfile_clauses adds or modifies a datafile or tempfile.

ADD DATAFILE | TEMPFILE	Adds to the tablespace a datafile or tempfile specified by *filespec* (see "filespec" on page 7-490).
	You can add a datafile or tempfile to a locally managed tablespace that is online or to a dictionary managed tablespace that is online or offline. Be sure the file is not in use by another database.

Note: As the syntax shows, you cannot combine an ADD clause with any other clauses in the same ALTER TABLESPACE statement. In addition, for a locally managed temporary tablespace, you cannot specify any of the other clauses for this tablespace at any time.

RENAME DATAFILE	renames one or more of the tablespace's datafiles. Take the tablespace offline before renaming the datafile. Each '*filename*' must fully specify a datafile using the conventions for filenames on your operating system.
	This clause merely associates the tablespace with the new file rather than the old one. This clause does not actually change the name of the operating system file. You must change the name of the file through your operating system.

autoextend_clause enables or disables the autoextending of the size of the datafile in the tablespace.

OFF	disables autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER TABLESPACE AUTOEXTEND statements.
ON	enables autoextend.
NEXT	specifies the size in bytes of the next increment of disk space to be allocated automatically to the datafile when more extents are required. Use K or M to specify this size in kilobytes or megabytes. The default is one data block.
maxsize_clause	specifies maximum disk space allowed for automatic extension of the datafile.
	UNLIMITED sets no limit on allocating disk space to the datafile.

DEFAULT *storage_clause* — specifies the new default storage parameters for objects subsequently created in the tablespace. For a dictionary-managed temporary table, Oracle considers only the NEXT parameter of the *storage_clause*. See the "storage_clause" on page 7-575.

Restriction: You cannot specify this clause for a locally managed tablespace.

MINIMUM EXTENT *integer* — controls free space fragmentation in the tablespace by ensuring that every used or free extent size in a tablespace is at least as large as, and is a multiple of, *integer*. This clause is not relevant for a dictionary-managed temporary tablespace. For more information about using MINIMUM EXTENT to control space fragmentation, see *Oracle8i Administrator's Guide*.

Restriction: You cannot specify this clause for a locally managed tablespace.

ONLINE — brings the tablespace online.

OFFLINE — takes the tablespace offline and prevents further access to its segments.

- NORMAL — flushes all blocks in all datafiles in the tablespace out of the SGA. You need not perform media recovery on this tablespace before bringing it back online. This is the default.
- TEMPORARY — performs a checkpoint for all online datafiles in the tablespace but does not ensure that all files can be written. Any offline files may require media recovery before you bring the tablespace back online.
- IMMEDIATE — does not ensure that tablespace files are available and does not perform a checkpoint. You must perform media recovery on the tablespace before bringing it back online.
- FOR RECOVER — takes the production database tablespaces in the recovery set offline for tablespace point-in-time recovery. For additional information see *Oracle8i Backup and Recovery Guide*.

Suggestion: Before taking a tablespace offline for a long time, you may want to alter the tablespace allocation of any users who have been assigned the tablespace as either a default or temporary tablespace. When the tablespace is offline, these users cannot allocate space for objects or sort areas in the tablespace. For more information, see "ALTER USER" on page 7-179.

BEGIN BACKUP — signifies that an open backup is to be performed on the datafiles that make up this tablespace. This clause does not prevent users from accessing the tablespace. You must use this clause before beginning an open backup. You cannot use this clause on a read-only tablespace.

Note: While the backup is in progress, you cannot take the tablespace offline normally, shut down the instance, or begin another backup of the tablespace.

END BACKUP — signifies that an open backup of the tablespace is complete. Use this clause as soon as possible after completing an open backup. You cannot use this clause on a read-only tablespace.

	If you forget to indicate the end of an online tablespace backup, and an instance failure or SHUTDOWN ABORT occurs, Oracle assumes that media recovery (possibly requiring archived redo log) is necessary at the next instance start up. To restart the database without media recovery, see *Oracle8i Administrator's Guide*.
READ ONLY	signifies that no further write operations are allowed on the tablespace. (This clause waits for all existing transactions either to commit or roll back before taking effect.) The tablespace becomes read only. Once a tablespace is read only, you can copy its files to read-only media. You must then rename the datafiles in the control file to point to the new location by using the SQL statement ALTER DATABASE ... RENAME. See "ALTER DATABASE" on page 7-6. For more information on read-only tablespaces, see *Oracle8i Concepts*.
READ WRITE	signifies that write operations are allowed on a previously read-only tablespace.
PERMANENT	specifies that the tablespace is to be converted from a temporary to a permanent one. A permanent tablespace is one in which permanent database objects can be stored. This is the default when a tablespace is created.
TEMPORARY	specifies that the tablespace is to be converted from a permanent to a temporary one. A temporary tablespace is one in which no permanent database objects can be stored. Objects in a temporary tablespace persist only for the duration of the session.
COALESCE	for each datafile in the tablespace, coalesces all contiguous free extents into larger contiguous extents. **Restriction:** COALESCE cannot be specified with any other statement clause.

Examples

Backup Examples The following statement signals to the database that a backup is about to begin:

```
ALTER TABLESPACE accounting
    BEGIN BACKUP;
```

The following statement signals to the database that the backup is finished:

```
ALTER TABLESPACE accounting
    END BACKUP;
```

Moving and Renaming Example This example moves and renames a datafile associated with the ACCOUNTING tablespace from 'DISKA:PAY1.DAT' to 'DISKB:RECEIVE1.DAT':

1. Take the tablespace offline using an ALTER TABLESPACE statement with the OFFLINE clause:

```
ALTER TABLESPACE accounting OFFLINE NORMAL;
```

2. Copy the file from 'DISKA:PAY1.DAT' to 'DISKB:RECEIVE1.DAT' using your operating system's commands.
3. Rename the datafile using the ALTER TABLESPACE statement with the RENAME DATAFILE clause:

```
ALTER TABLESPACE accounting
  RENAME DATAFILE 'diska:pay1.dbf'
  TO     'diskb:receive1.dbf';
```

4. Bring the tablespace back online using an ALTER TABLESPACE statement with the ONLINE clause:

```
ALTER TABLESPACE accounting ONLINE;
```

Adding a Datafile Example The following statement adds a datafile to the tablespace and changes the default logging attribute to NOLOGGING. When more space is needed, new extents of size 10 kilobytes will be added up to a maximum of 100 kilobytes:

```
ALTER TABLESPACE accounting NOLOGGING
    ADD DATAFILE 'disk3:pay3.dbf'
    SIZE 50K
    AUTOEXTEND ON
    NEXT 10K
    MAXSIZE 100K;
```

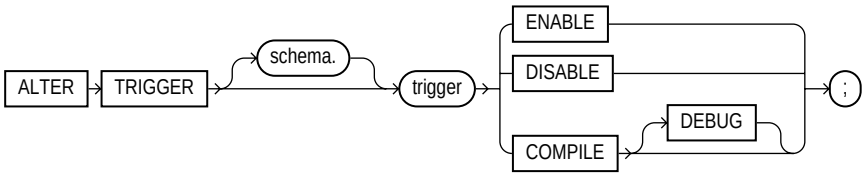
Altering a tablespace logging attribute has no affect on the logging attributes of the existing schema objects within the tablespace. The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

Changing Extent Allocation Example The following statement changes the allocation of every extent of TABSPACE_ST to a multiple of 128K:

```
ALTER TABLESPACE tabspace_st MINIMUM EXTENT 128K;
```

ALTER TRIGGER

Syntax



Purpose

To enable, disable, or compile a database trigger. For information on creating a trigger, see "CREATE TRIGGER" on page 7-401. For information on dropping a trigger, see "DROP TRIGGER" on page 7-479.

> **Note:** This statement does not change the declaration or definition of an existing trigger. To redeclare or redefine a trigger, use the CREATE TRIGGER statement with OR REPLACE.

Prerequisites

The trigger must be in your own schema or you must have ALTER ANY TRIGGER system privilege.

In addition, to alter a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER system privilege. For more information on triggers based on DATABASE, see "CREATE TRIGGER" on page 7-401.

Keywords and Parameters

schema	is the schema containing the trigger. If you omit *schema*, Oracle assumes the trigger is in your own schema.
trigger	is the name of the trigger to be altered.
ENABLE	enables the trigger. You can also use the ENABLE ALL TRIGGERS clause of ALTER TABLE to enable all triggers associated with a table. See "ALTER TABLE" on page 7-113.

DISABLE	disables the trigger. You can also use the DISABLE ALL TRIGGERS clause of ALTER TABLE to disable all triggers associated with a table. See "ALTER TABLE" on page 7-113.
COMPILE	explicitly compiles the trigger, whether it is valid or invalid. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead. Oracle first recompiles objects upon which the trigger depends, if any of these objects are invalid. If Oracle recompiles the trigger successfully, the trigger becomes valid. If recompiling the trigger results in compilation errors, then Oracle returns an error and the trigger remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS. For information on debugging procedures, see *Oracle8i Application Developer's Guide - Fundamentals*. For information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8i Concepts*. DEBUG — instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. This clause can be used for normal triggers and for instead-of triggers.

Examples

Consider a trigger named REORDER created on the INVENTORY table. The trigger is fired whenever an UPDATE statement reduces the number of a particular part on hand below the part's reorder point. The trigger inserts into a table of pending orders a row that contains the part number, a reorder quantity, and the current date.

When this trigger is created, Oracle enables it automatically. You can subsequently disable the trigger with the following statement:

```
ALTER TRIGGER reorder DISABLE;
```

When the trigger is disabled, Oracle does not fire the trigger when an UPDATE statement causes the part's inventory to fall below its reorder point.

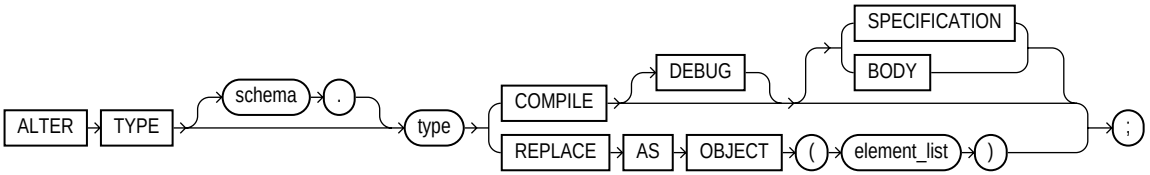
After disabling the trigger, you can subsequently enable it with the following statement:

```
ALTER TRIGGER reorder ENABLE;
```

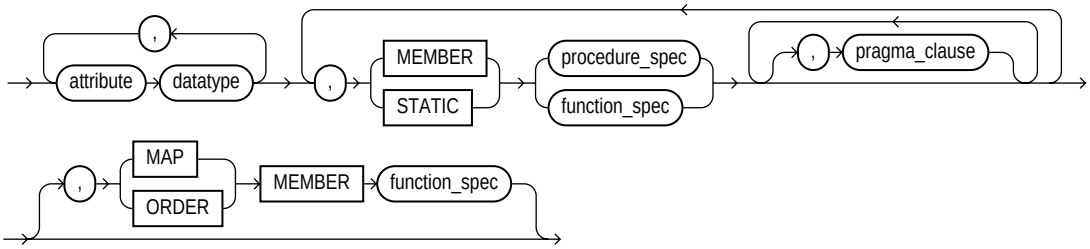
After you reenable the trigger, Oracle fires the trigger whenever a part's inventory falls below its reorder point as a result of an UPDATE statement. It is possible that a part's inventory falls below its reorder point while the trigger was disabled. In that case, when you reenable the trigger, Oracle does not automatically fire the trigger for this part until another transaction further reduces the inventory.

ALTER TYPE

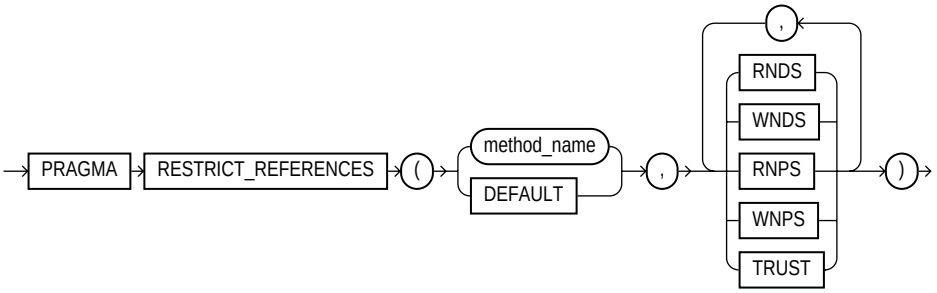
Syntax



element_list::=



pragma_clause::=



Purpose

To recompile the specification and / or body, or to change the specification of an object type by adding new object member subprogram specifications.

You cannot change the existing properties (attributes, member subprograms, map or order functions) of an object type, but you can add new member subprogram specifications.

Prerequisites

The object type must be in your own schema and you must have CREATE TYPE or CREATE ANY TYPE system privilege, or you must have ALTER ANY TYPE system privileges.

Keywords and Parameters

schema	is the schema that contains the type. If you omit *schema*, Oracle assumes the type is in your current schema.
type	is the name of an object type, a nested table type, or a rowid type.
COMPILE	compiles the object type specification and body. This is the default if neither SPECIFICATION nor BODY is specified. If recompiling the type results in compilation errors, then Oracle returns an error and the type remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS. SPECIFICATION compiles only the object type specification. BODY compiles only the object type body.
DEBUG	instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger.
REPLACE AS OBJECT	adds new member subprogram specifications. This clause is valid only for object types, not for nested table or varray types.
attribute	is an object attribute name. Attributes are data items with a name and a type specifier that form the structure of the object.
MEMBER | STATIC	specifies a function or procedure subprogram associated with the object type which is referenced as an attribute. For a description of the difference between member and static methods, and for examples, see "CREATE TYPE" on page 7-411. For information about overloading subprogram names within a package, see the *PL/SQL User's Guide and Reference*. You must specify a corresponding method body in the object type body for each procedure or function specification. See "CREATE TYPE BODY" on page 7-421. *procedure_spec* is the specification of a procedure subprogram. *function_spec* is the specification of a function subprogram.

pragma_clause is a complier directive that denies member functions read / write access to database tables, packaged variables, or both, and thereby helps to avoid side effects. For more information, see *Oracle8i Application Developer's Guide - Fundamentals*.

method	is the name of the MEMBER function or procedure to which the pragma is being applied.
DEFAULT	specifies that the pragma should be applied to all methods in the type for which a pragma has not been explicitly specified.
WNDS	specifies the constraint *writes no database state* (does not modify database tables).
WNPS	specifies the constraint *writes no package state* (does not modify packaged variables).
RNDS	specifies the constraint *reads no database state* (does not query database tables).
RNPS	specifies the constraint *reads no package state* (does not reference package variables).
TRUST	specifies that the restrictions listed in the pragma are not actually to be enforced, but are simply trusted to be true.

MAP | ORDER MEMBER *function_spec*

MAP specifies a member function (MAP method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined *scalar* type. Oracle uses the ordering for comparison operators and ORDER BY clauses.

If the argument to the map method is null, the map method returns null and the method is not invoked.

An object specification can contain only one map method, which must be a function. The result type must be a predefined SQL scalar type, and the map function can have no arguments other than the implicit SELF argument.

Note: If *type_name* will be referenced in queries involving sorts (through ORDER BY, GROUP BY, DISTINCT, or UNION clauses) or joins, and you want those queries to be parallelized, you must specify a MAP member function.

ORDER specifies a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, zero, or positive indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

If either argument to the order method is null, the order method returns null and the method is not invoked.

When instances of the same object type definition are compared in an ORDER BY clause, the order method function is invoked.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

You can declare either a MAP method or an ORDER method, but not both. If you declare either method, you can compare object instances in SQL.

If you do not declare either method, you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method needs to be specified to determine the equality of two object types. For more information about object value comparisons, see "Object Values" on page 2-30.

Examples

Adding a Member Function In the following example, member function QTR is added to the type definition of DATA_T.

```
CREATE TYPE data_t AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION prod(invent NUMBER) RETURN NUMBER
   );

   CREATE TYPE BODY data_t IS
      MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
         BEGIN
             RETURN (year + invent);
         END;
      END;

   ALTER TYPE data_t REPLACE AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION  prod(invent NUMBER) RETURN NUMBER,
     MEMBER FUNCTION  qtr(der_qtr DATE) RETURN CHAR
   );

   CREATE OR REPLACE TYPE BODY data_t IS
      MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
      MEMBER FUNCTION qtr(der_qtr DATE) RETURN CHAR IS
         BEGIN
            RETURN (year + invent);
```

```
      END;
      BEGIN
         RETURN 'FIRST';
      END;
   END;
```

Recompiling a Type The following example creates and then recompiles type LOAN_T:

```
CREATE TYPE loan_t AS OBJECT
  ( loan_num        NUMBER,
    interest_rate   FLOAT,
    amount          FLOAT,
    start_date      DATE,
    end_date        DATE );

ALTER TYPE loan_t COMPILE;
```

Recompiling a Type Body The following example compiles the type body of LINK2.

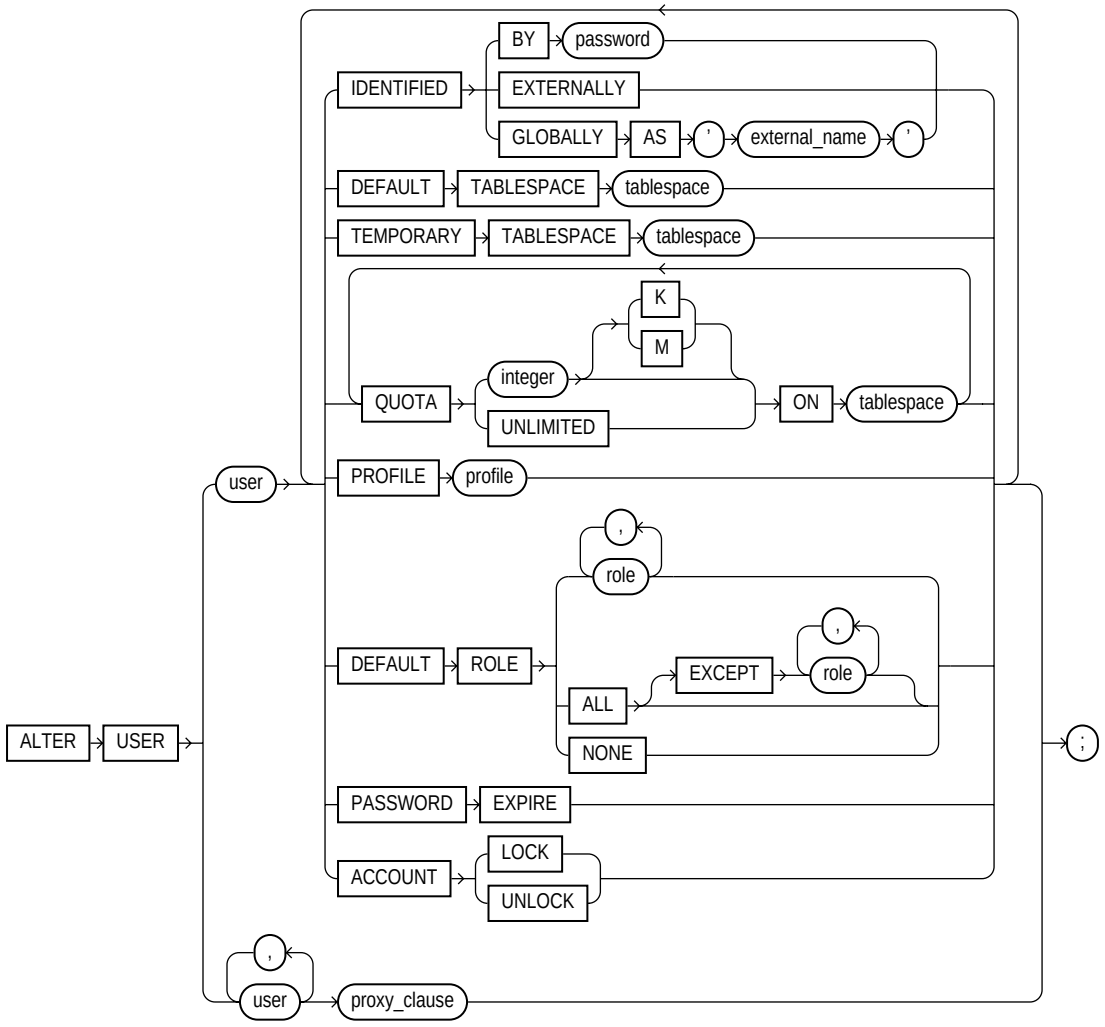
```
CREATE TYPE link1 AS OBJECT
  (a NUMBER);

CREATE TYPE link2 AS OBJECT
  (a NUMBER,
   b link1,
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);

CREATE TYPE BODY link2 AS
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS t13 link1;
      BEGIN t13 := link1(13);
         dbms_output.put_line(t13.a);
         RETURN 5;
      END;
   END;

CREATE TYPE link3 AS OBJECT (a link2);
CREATE TYPE link4 AS OBJECT (a link3);
CREATE TYPE link5 AS OBJECT (a link4);
ALTER TYPE link2 COMPILE BODY;
```

Recompiling a Type Specification The following example compiles the type specification of LINK2.

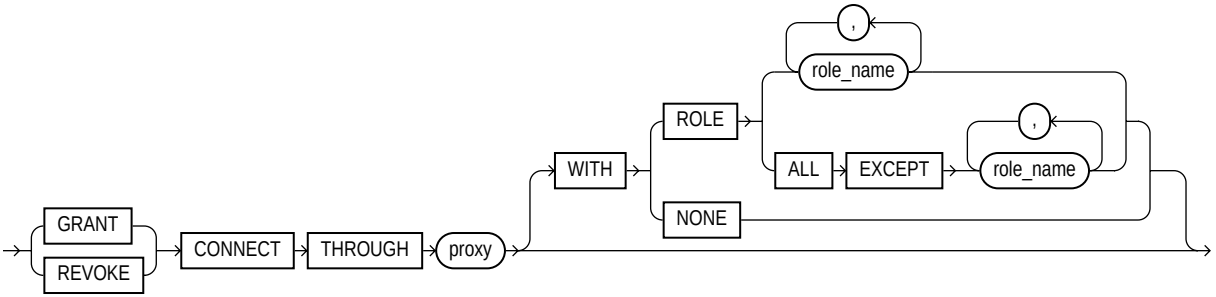
```
CREATE TYPE link1 AS OBJECT
  (a NUMBER);

CREATE TYPE link2 AS OBJECT
  (a NUMBER,
   b link1,
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);

CREATE TYPE BODY link2 AS
    MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS t14 link1;
      BEGIN t14 := link1(14);
         dbms_output.put_line(t14.a);
         RETURN 5;
      END;
    END;

CREATE TYPE link3 AS OBJECT (a link2);
CREATE TYPE link4 AS OBJECT (a link3);
CREATE TYPE link5 AS OBJECT (a link4);
ALTER TYPE link2 COMPILE SPECIFICATION;
```

ALTER USER

Syntax



proxy_clause::=



Purpose

To change the authentication or database resource characteristics of a database user.

To permit a proxy server to connect as a client without authentication.

> **Note:** ALTER USER syntax does not accept the old password. Therefore it neither authenticates using the old password nor checks the new password against the old before setting the new password. If these checks against the old password are important, use the OCIPasswordChange() call instead of ALTER USER. For more information, see *Oracle Call Interface Programmer's Guide*.

Prerequisites

You must have the ALTER USER system privilege. However, you can change your own password without this privilege.

Keywords and Parameters

The keywords and parameters shown below are unique to ALTER USER or have different functionality than they have in CREATE USER. All the remaining keywords and parameters in the ALTER USER statement have the same meaning as in the CREATE USER statement. For information on these keywords and parameters, see "CREATE USER" on page 7-425.

To assign limits on database resources to a user, see "CREATE PROFILE" on page 7-338.

IDENTIFIED GLOBALLY AS	indicates that a user must be authenticated by way of an LDAP V3 compliant directory service such as Oracle Internet Directory. (See also "CREATE USER" on page 7-425.) You can change a user's access verification method to IDENTIFIED GLOBALLY AS '*external_name*' only if all external roles granted directly to the user are revoked. You can change a user created as IDENTIFIED GLOBALLY AS '*external_name*' to IDENTIFIED BY *password* or IDENTIFIED EXTERNALLY.
DEFAULT ROLE	can contain only roles that have been granted directly to the user with a GRANT statement. You cannot use the DEFAULT ROLE clause to enable: ■ roles not granted to the user ■ roles granted through other roles ■ roles managed by an external service (such as the operating system), or by the Oracle Internet Directory Oracle enables default roles at logon without requiring the user to specify their passwords. For more information on roles, see "CREATE ROLE" on page 7-344.
proxy_clause	controls the ability of a **proxy** (an application or application server) to connect as the specified user and to activate all, some, or none of the user's roles. For more information on proxies and their use of the database, see *Oracle8i Concepts*.

GRANT	allows the connection.
REVOKE	prohibits the connection.
proxy	identifies the proxy connecting to Oracle.
WITH ROLE	specifies the roles that the application is permitted to activate after it connects as the user. If you do not include this clause, Oracle activates all roles granted to the specified user automatically.
role_name	permits the proxy to connect as the specified user and to activate only the roles that are specified by *role_name*.
ALL EXCEPT *role_name*	permits the proxy to connect as the specified user and to activate all roles associated with that user except those specified by *role_name*.
NONE	permits the proxy to connect as the specified user, but prohibits the proxy from activating any of that user's roles after connecting.

Examples

General Examples The following statement changes the user SCOTT's password to LION and default tablespace to the tablespace TSTEST:

```
ALTER USER scott
    IDENTIFIED BY lion
```

```
    DEFAULT TABLESPACE tstest;
```

The following statement assigns the CLERK profile to SCOTT:

```
ALTER USER scott
    PROFILE clerk;
```

In subsequent sessions, SCOTT is restricted by limits in the CLERK profile.

The following statement makes all roles granted directly to SCOTT default roles, except the AGENT role:

```
ALTER USER scott
    DEFAULT ROLE ALL EXCEPT agent;
```

At the beginning of SCOTT's next session, Oracle enables all roles granted directly to SCOTT except the AGENT role.

Authentication Examples The following statement changes user TOM's authentication mechanism:

```
ALTER USER tom IDENTIFIED GLOBALLY AS 'CN=tom,O=oracle,C=US';
```

The following statement causes user FRED's password to expire:

```
ALTER USER fred PASSWORD EXPIRE;
```

If you cause a database user's password to expire with PASSWORD EXPIRE, the user (or the DBA) must change the password before attempting to log in to the database following the expiration. However, tools such as SQL*Plus allow you to change the password on the first attempted login following the expiration.

Proxy Examples The following statement permits the proxy user APPSERVER1 to connect as the user JANE. It also allows APPSERVER1 to activate the role INVENTORY:

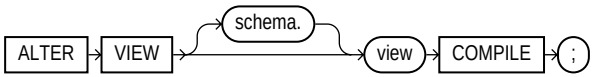
```
ALTER USER jane GRANT CONNECT THROUGH appserver1 WITH ROLE inventory;
```

The following statement takes away the right of proxy user APPSERVER1 to connect as the user JANE:

```
ALTER USER jane REVOKE CONNECT THROUGH appserver1;
```

ALTER VIEW

Syntax



Purpose

To explicitly recompile a view that is invalid. Explicit recompilation allows you to locate recompilation errors before run time. You may want to recompile a view explicitly after altering one of its base tables to ensure that the alteration does not affect the view or other objects that depend on it.

When you issue an ALTER VIEW statement, Oracle recompiles the view regardless of whether it is valid or invalid. Oracle also invalidates any local objects that depend on the view. For more about dependencies among schema objects, see *Oracle8i Concepts*.

Notes:

- This statement does not change the definition of an existing view. To redefine a view, you must use CREATE VIEW with OR REPLACE. See "CREATE VIEW" on page 7-430.
- If you alter a view that is referenced by one or more materialized views, those materialized views are invalidated. Invalid materialized views cannot be used by query rewrite and cannot be refreshed. To revalidate an invalid materialized view, see "ALTER MATERIALIZED VIEW / SNAPSHOT" on page 7-45. For information on materialized views in general, see *Oracle8i Tuning*.

Prerequisites

The view must be in your own schema or you must have ALTER ANY TABLE system privilege.

Keywords and Parameters

schema	is the schema containing the view. If you omit *schema*, Oracle assumes the view is in your own schema.
view	is the name of the view to be recompiled.
COMPILE	causes Oracle to recompile the view. The COMPILE keyword is required.

Example

To recompile the view CUSTOMER_VIEW, issue the following statement:

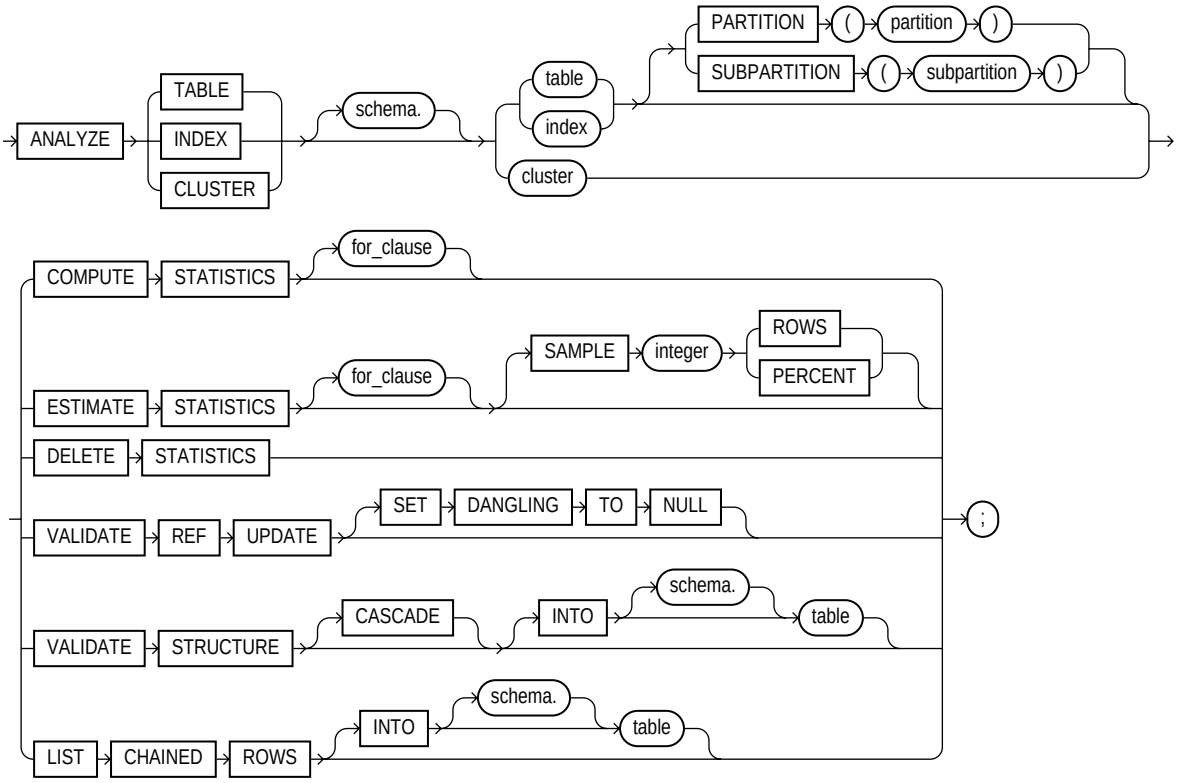
```
ALTER VIEW customer_view
    COMPILE;
```

If Oracle encounters no compilation errors while recompiling CUSTOMER_VIEW, CUSTOMER_VIEW becomes valid. If recompiling results in compilation errors, Oracle returns an error and CUSTOMER_VIEW remains invalid.

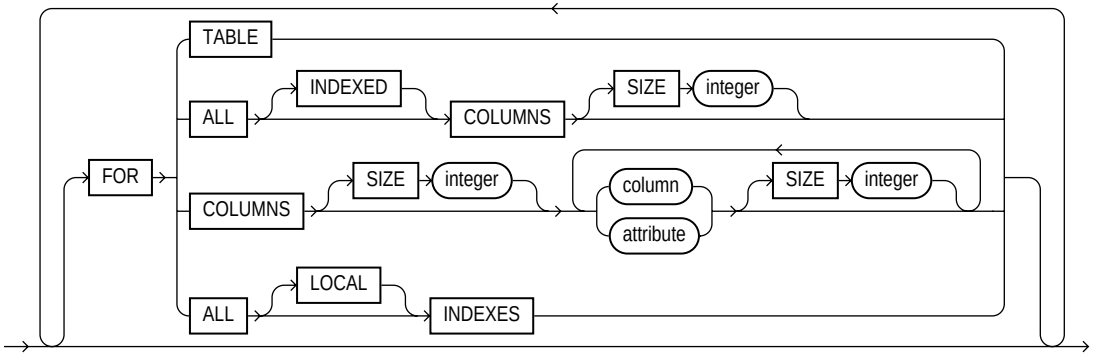
Oracle also invalidates all dependent objects. These objects include any procedures, functions, package bodies, and views that reference CUSTOMER_VIEW. If you subsequently reference one of these objects without first explicitly recompiling it, Oracle recompiles it implicitly at run time.

ANALYZE

Syntax



for_clause::=



Purpose

To collect or delete statistics about an index or index partition, table or table partition, index-organized table, cluster, or scalar object attribute.

To validate the structure of an index or index partition, table or table partition, index-organized table, cluster, or object reference (REF).

To identify migrated and chained rows of a table or cluster.

Prerequisites

The schema object to be analyzed must be in your own schema or you must have the ANALYZE ANY system privilege.

If you want to list chained rows of a table or cluster into a list table, the list table must be in your own schema, or you must have INSERT privilege on the list table, or you must have INSERT ANY TABLE system privilege.

If you want to validate a partitioned table, you must have INSERT privilege on the table into which you list analyzed rowids, or you must have INSERT ANY TABLE system privilege.

Keywords and Parameters

schema	is the schema containing the index, table, or cluster. If you omit *schema*, Oracle assumes the index, table, or cluster is in your own schema.
INDEX *index*	identifies an index to be analyzed (if no *for_clause* is used).

Oracle collects the following statistics for an index (statistics marked with an asterisk are always computed exactly):

- Depth of the index from its root block to its leaf blocks*
- Number of leaf blocks
- Number of distinct index values
- Average number of leaf blocks per index value
- Average number of data blocks per index value (for an index on a table)
- Clustering factor (how well ordered the rows are about the indexed values)

Index statistics appear in the data dictionary views USER_INDEXES, ALL_INDEXES, and DBA_INDEXES.

For a **domain index**, this statement invokes the user-defined statistics collection function specified in the statistics type associated with the index (see "ASSOCIATE STATISTICS" on page 7-194). If no statistics type is associated with the domain index, the statistics type associated with its indextype is used. If no statistics type exists for either the index or its indextype, no user-defined statistics are collected. User-defined index statistics appear in the data dictionary views USER_USTATS, ALL_USTATS, and DBA_USTATS.

Restriction: You cannot analyze a domain index that is marked LOADING or FAILED.

For more information on domain indexes, see "CREATE INDEX" on page 7-273.

TABLE *table* identifies a table to be analyzed. When you collect statistics for a table, Oracle also automatically collects the statistics for each of the table's indexes and domain indexes, provided that no *for_clauses* are used.

When you analyze a table, Oracle collects statistics about expressions occurring in any function-based indexes as well. Therefore, be sure to create function-based indexes on the table before analyzing the table. For more information about function-based indexes, see "CREATE INDEX" on page 7-273.

When analyzing a table, Oracle skips all domain indexes marked LOADING or FAILED.

Table statistics, including the status of domain indexes, appear in the data dictionary views USER_TABLES, ALL_TABLES, and DBA_TABLES.

Oracle collects the following statistics for a table (statistics marked with an asterisk are always computed exactly):

- Number of rows
- Number of data blocks below the high water mark (that is, the number of data blocks that have been formatted to receive data, regardless whether they currently contain data or are empty) *
- Number of data blocks allocated to the table that have never been used *
- Average available free space in each data block in bytes
- Number of chained rows
- Average row length, including the row's overhead, in bytes

Restrictions:

- You cannot use ANALYZE to collect statistics on data dictionary tables.
- You cannot use ANALYZE to collect default statistics on a temporary table. However, if you have created an association between one or more columns of a temporary table and a user-defined statistics type, you can use ANALYZE to collect the user-defined statistics on the temporary table. (The association must already exist.) For more information, see "ASSOCIATE STATISTICS" on page 7-194.
- You cannot compute or estimate statistics for the following column types: REFs, varrays, nested tables, LOBs (LOBs are not analyzed, they are skipped), LONGs, or object types. However, if a statistics type is associated with such a column, user-defined statistics are collected.

PARTITION | SUBPARTITION — specifies that statistics will be gathered for *partition* or *subpartition*. You cannot use this clause when analyzing clusters.

If you specify PARTITION and *table* is composite-partitioned, Oracle analyzes all the subpartitions within the specified partition.

CLUSTER *cluster* — identifies a cluster to be analyzed. When you collect statistics for a cluster, Oracle also automatically collects the statistics for all the cluster's tables and all their indexes, including the cluster index.

- For an indexed cluster, Oracle collects the average number of data blocks taken up by a single cluster key value and all of its rows.
- For a hash cluster, Oracle collects the average number of data blocks taken up by a single hash key value and all of its rows.

These statistics appear in the data dictionary views USER_CLUSTERS and DBA_CLUSTERS.

COMPUTE STATISTICS — computes exact statistics about the analyzed object and stores them in the data dictionary. When you analyze a table, both table and column statistics are collected.

ESTIMATE STATISTICS — estimates statistics about the analyzed object and stores them in the data dictionary.

Both computed and estimated statistics are used by the Oracle optimizer to choose the execution plan for SQL statements that access analyzed objects. These statistics may also be useful to application developers who write such statements. For information on how these statistics are used, see *Oracle8i Tuning*.

SAMPLE *integer* specifies the amount of data from the analyzed object Oracle samples to estimate statistics. If you omit this parameter, Oracle samples 1064 rows.

The default sample value is adequate for tables up to a few thousand rows. If your tables are larger, specify a higher value for SAMPLE. If you specify more than half of the data, Oracle reads all the data and computes the statistics.

ROWS causes Oracle to sample *integer* rows of the table or cluster or *integer* entries from the index. The integer must be at least 1.

PERCENT causes Oracle to sample *integer* percent of the rows from the table or cluster or *integer* percent of the index entries. The integer can range from 1 to 99.

for_clause specifies whether an entire table or index, or just particular columns, will be analyzed. The following clauses apply only to the ANALYZE TABLE version of this statement:

- FOR TABLE restricts the statistics collected to only table statistics rather than table and column statistics.
- FOR COLUMNS restricts the statistics collected to only column statistics for the specified columns and scalar object attributes, rather than for all columns and attributes.
- FOR ALL COLUMNS collects column statistics for all columns and scalar object attributes.
- FOR ALL INDEXED COLUMNS collects column statistics for all indexed columns in the table.

  Column statistics can be based on the entire column or can use a histogram by specifying SIZE (see below). For more information on histograms, see *Oracle8i Tuning*. See also "Histogram Examples" on page 7-192.

  Oracle collects the following column statistics:

  - Number of distinct values in the column as a whole

  - Maximum and minimum values in each band

  Column statistics appear in the data dictionary views USER_TAB_COLUMNS, ALL_TAB_COLUMNS, and DBA_TAB_COLUMNS. Histograms appear in the data dictionary views USER_HISTOGRAMS, DBA_HISTOGRAMS, and ALL_HISTOGRAMS.

Note: The MAXVALUE and MINVALUE columns of USER_, DBA_, and ALL_TAB_ COLUMNS have a length of 32 bytes. If you analyze columns with a length >32 bytes, and if the columns are padded with leading blanks, Oracle may take into account only the leading blanks and return unexpected statistics.

If a user-defined statistics type has been associated with any columns, the *for_clause* collects user-defined statistics using that statistics type. If no statistics type is associated with a column, Oracle checks to see if any statistics type has been associated with the type of the column, and uses that statistics type. If no statistics type has been associated with either the column or its user-defined type, no user-defined statistics are collected. User-defined column statistics appear in the data dictionary views USER_USTATS, ALL_USTATS, and DBA_USTATS.

Note: If you want to collect statistics on both the table as a whole and on one or more columns, be sure to generate the statistics for the table first, and then for the columns. Otherwise, the table-only ANALYZE will overwrite the histograms generated by the column ANALYZE. For example, issue the following statements:

```
ANALYZE TABLE emp ESTIMATE STATISTICS;

ANALYZE TABLE emp ESTIMATE STATISTICS FOR ALL COLUMNS;
```

- *attribute* specifies the qualified column name of an item in an object.
- FOR ALL INDEXES specifies that all indexes associated with the table will be analyzed.
- FOR ALL LOCAL INDEXES specifies that all local index partitions are analyzed. You must specify the keyword LOCAL if the PARTITION clause and INDEX are specified.
- SIZE specifies the maximum number of partitions in the histogram. The default value is 75, minimum value is 1, and maximum value is 254.

DELETE STATISTICS — deletes any statistics about the analyzed object that are currently stored in the data dictionary. Use this statement when you no longer want Oracle to use the statistics.

When you use this clause on a table, Oracle also automatically removes statistics for all the table's indexes. When you use this clause on a cluster, Oracle also automatically removes statistics for all the cluster's tables and all their indexes, including the cluster index.

If user-defined column or index statistics were collected for an object, Oracle also removes the user-defined statistics by invoking the statistics deletion function specified in the statistics type that was used to collect the statistics.

VALIDATE REF UPDATE — validates the REFs in the specified table, checks the rowid portion in each REF, compares it with the true rowid, and corrects, if necessary. You can use this clause only when analyzing a table.

SET DANGLING TO NULL — sets to NULL any REFs (whether or not scoped) in the specified table that are found to point to an invalid or nonexistent object.

Note: If the owner of the table does not have SELECT object privilege on the referenced objects, Oracle will consider them invalid and set them to NULL. Subsequently these REFs will not be available in a query, even if it is issued by user with appropriate privileges on the objects.

VALIDATE STRUCTURE

validates the structure of the analyzed object. The statistics collected by this clause are not used by the Oracle optimizer, as are statistics collected by the COMPUTE STATISTICS and ESTIMATE STATISTICS clauses.

- For a table, Oracle verifies the integrity of each of the table's data blocks and rows.
- For a cluster, Oracle automatically validates the structure of the cluster's tables.
- For a partitioned table, Oracle also verifies that the row belongs to the correct partition. If the row does not collate correctly, the rowid is inserted into the INVALID_ROWS table.
- For a temporary table, Oracle validates the structure of the table and its indexes during the current session.
- For an index, Oracle verifies the integrity of each data block in the index and checks for block corruption. This clause does not confirm that each row in the table has an index entry or that each index entry points to a row in the table. You can perform these operations by validating the structure of the table with the CASCADE clause.

  Oracle stores statistics about the index in the data dictionary views INDEX_STATS and INDEX_HISTOGRAM, which are described in *Oracle8i Reference*.

Validating the structure of an object prevents SELECT, INSERT, UPDATE, and DELETE statements from concurrently accessing the object. Therefore, do not use this clause on the tables, clusters, and indexes of your production applications during periods of high database activity.

If Oracle encounters corruption in the structure of the object, an error message is returned to you. In this case, drop and re-create the object.

INTO

specifies a table into which Oracle lists the rowids of the partitions whose rows do not collate correctly. If you omit *schema*, Oracle assumes the list is in your own schema. If you omit this clause altogether, Oracle assumes that the table is named INVALID_ROWS. The SQL script used to create this table is UTLVALID.SQL.

CASCADE

validates the structure of the indexes associated with the table or cluster. If you use this clause when validating a table, Oracle also validates the table's indexes. If you use this clause when validating a cluster, Oracle also validates all the clustered tables' indexes, including the cluster index.

If you use this clause to validate an enabled (but previously disabled) function-based index, validation errors may result. In this case, you must rebuild the index.

LIST CHAINED ROWS	identifies migrated and chained rows of the analyzed table or cluster. You cannot use this clause when analyzing an index.
INTO	specifies a table into which Oracle lists the migrated and chained rows. If you omit *schema*, Oracle assumes the list table is in your own schema. If you omit this clause altogether, Oracle assumes that the table is named CHAINED_ROWS. The script used to create this table is UTLCHAIN1.SQL. The list table must be on your local database.

Note: You can use the UTLCHAIN1.SQL script with index-organized tables. You could not use earlier versions of the script for this purpose. See *Oracle8i Migration* for compatibility information.

To analyze index-organized tables, you must create a separate chained-rows table for each index-organized table to accommodate the primary-key storage of index-organized tables. Use the SQL scripts DBMSIOTC.SQL and PRVTIOTC.PLB to define the BUILD_CHAIN_ROWS_TABLE procedure, and then execute this procedure to create an IOT_CHAINED_ROWS table for an index-organized table.

For information on the SQL scripts, see the DBMS_IOT package in *Oracle8i Supplied Packages Reference*. For information on eliminating migrated and chained rows, see *Oracle8i Tuning*.

Examples

Analyzing a Cluster The following statement estimates statistics for the CUST_HISTORY table and all of its indexes:

```
ANALYZE TABLE cust_history
   ESTIMATE STATISTICS;
```

Deleting Statistics The following statement deletes statistics about the CUST_HISTORY table and all its indexes from the data dictionary:

```
ANALYZE TABLE cust_history
   DELETE STATISTICS;
```

Histogram Examples The following statement creates a 10-band histogram on the SAL column of the EMP table:

```
ANALYZE TABLE emp
   COMPUTE STATISTICS FOR COLUMNS sal SIZE 10;
```

You can also collect histograms for a single partition of a table. The following statement analyzes the EMP table partition P1:

```
ANALYZE TABLE emp PARTITION (p1) COMPUTE STATISTICS;
```

Index Example The following statement validates the structure of the index PARTS_INDEX:

```
ANALYZE INDEX parts_index VALIDATE STRUCTURE;
```

Table Examples The following statement analyzes the EMP table and all of its indexes:

```
ANALYZE TABLE emp VALIDATE STRUCTURE CASCADE;
```

For a table, the VALIDATE REF UPDATE clause verifies the REFs in the specified table, checks the rowid portion of each REF, and then compares it with the true rowid. If the result is an incorrect rowid, the REF is updated so that the rowid portion is correct.

The following statement validates the REFs in the EMP table:

```
ANALYZE TABLE emp
   VALIDATE REF UPDATE;
```

Cluster Example The following statement analyzes the ORDER_CUSTS cluster, all of its tables, and all of their indexes, including the cluster index:

```
ANALYZE CLUSTER order_custs
    VALIDATE STRUCTURE CASCADE;
```

Chained Rows Example The following statement collects information about all the chained rows of the table ORDER_HIST:

```
ANALYZE TABLE order_hist
    LIST CHAINED ROWS INTO cr;
```

The preceding statement places the information into the table CR. You can then examine the rows with this query:

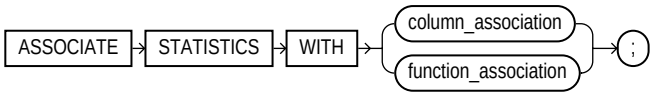
```
SELECT *
    FROM cr;

OWNER_NAME  TABLE_NAME  CLUSTER_NAME  HEAD_ROWID         TIMESTAMP
----------  ----------  ------------  ------------------ ---------
SCOTT       ORDER_HIST                AAAAZzAABAAABrXAAA 15-MAR-96
```

COMPUTE Example The following statement calculates statistics for a scalar object attribute:

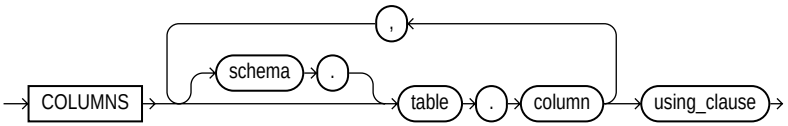
```
ANALYZE TABLE emp COMPUTE STATISTICS FOR COLUMNS addr.street;
```

ASSOCIATE STATISTICS

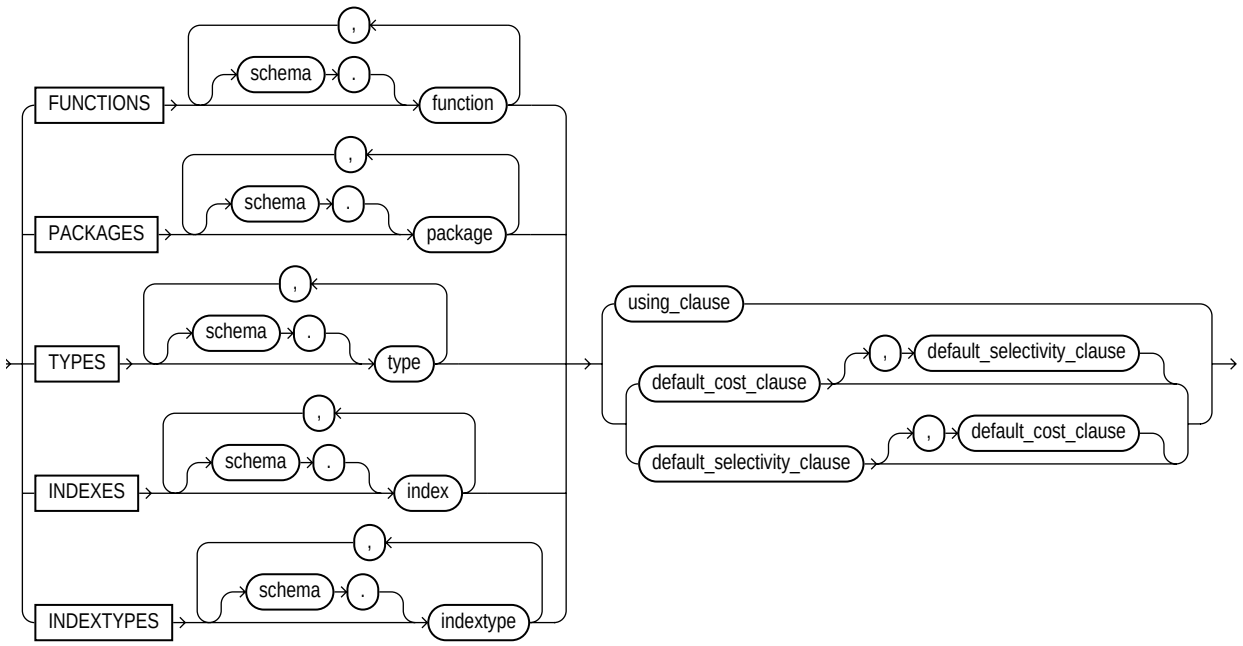
Syntax



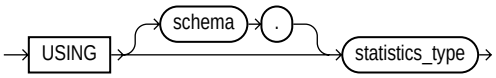
column_association::=



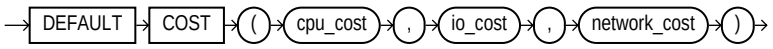
function_association::=



using_clause::=



default_cost_clause::=



default_selectivity_clause::=

Purpose

To associate a statistics type (or default statistics) containing functions relevant to statistics collection, selectivity, or cost with one or more columns, standalone functions, packages, types, domain indexes, or indextypes.

For a listing of all current statistics type associations, refer to the USER_ASSOCIATIONS table. If you analyze the object with which you are associating statistics, you can also view the associations in the USER_USTATS table. For information on the order of precedence with which ANALYZE uses associations, see "ANALYZE" on page 7-185.

Prerequisites

To issue this statement, you must have the appropriate privileges to alter the base object (table, function, package, type, domain index, or indextype). In addition, unless you are associating only default statistics, you must have execute privilege on the statistics type. The statistics type must already have been defined. For information on defining types, see "CREATE TYPE" on page 7-411.

Keywords and Parameters

column_association specifies a list of one or more table columns. If you do not specify *schema*, Oracle assumes the table is in your own schema.

function_association	specifies a list of one or more standalone functions, packages, user-defined datatypes, domain indexes, or indextypes. If you do not specify *schema*, Oracle assumes the object is in your own schema. ■ FUNCTIONS refers only to standalone functions, not to method types or to built-in functions. ■ TYPES refers only to user-defined types, not to internal SQL datatypes. **Restriction:** You cannot specify an object for which you have already defined an association. You must first disassociate the statistics from this object. See "DISASSOCIATE STATISTICS" on page 7-444.
using_clause	specifies the statistics type being associated with columns, functions, packages, types, domain indexes, or indextypes. The *statistics_type* must already have been created.
default_cost_clause	specifies default costs for standalone functions, packages, types, domain indexes, or indextypes. If you specify this clause, you must include one number each for CPU cost, I/O cost, and network cost, in that order. Each cost is for a single execution of the function or method or for a single domain index access. Accepted values are integers of zero or greater.
default_selectivity_clause	specifies as a percent the default selectivity for predicates with standalone functions, types, packages, or user-defined operators. The *default_selectivity* must be a whole number between 0 and 100. Values outside this range are ignored. **Restriction:** You cannot specify DEFAULT SELECTIVITY for domain indexes or indextypes.

Examples

Standalone Function Example This statement creates an association for a standalone function FN and causes the optimizer to call the appropriate cost function (if present) in the statistics type STAT_FN.

```
ASSOCIATE STATISTICS WITH FUNCTIONS fn USING stat_fn;
```

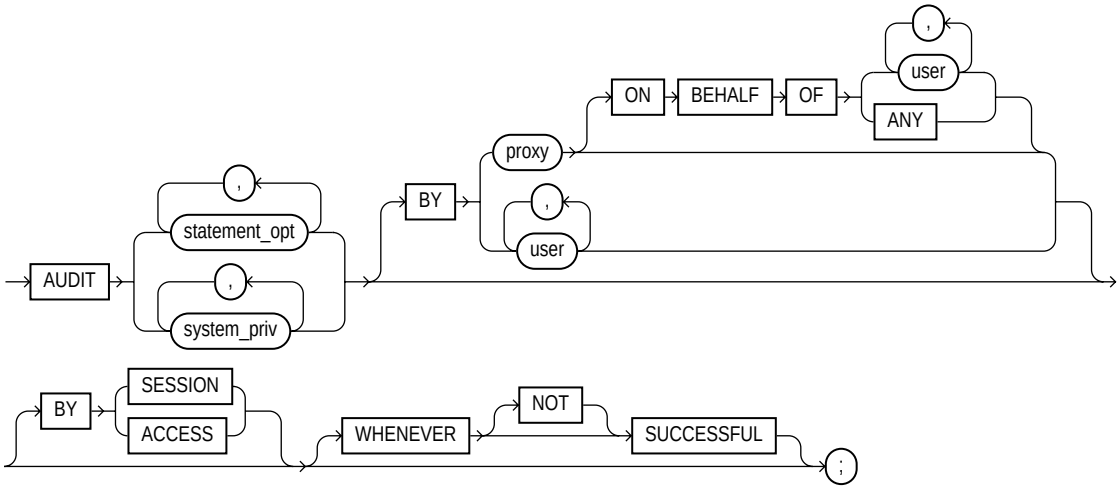
Default Cost Example This statement specifies that using the domain index T_A to implement a given predicate always has a CPU cost of 100, I/O of 5, and network cost of 0.

```
ASSOCIATE STATISTICS WITH INDEXES t_a DEFAULT COST (100,5,0);
```

The optimizer will simply use these default costs instead of calling a cost function.

AUDIT *sql_statements*

Syntax



Purpose

To track the occurrence of specific SQL statements in subsequent user sessions. Auditing options specified by the AUDIT *sql_statements* statement apply only to subsequent sessions, not to current sessions.

To choose particular schema objects for auditing, see "AUDIT schema_objects" on page 7-205. For information on disabling auditing of SQL statements, see "NOAUDIT sql_statements" on page 7-523.

Prerequisites

You must have AUDIT SYSTEM system privilege.

You must enable auditing by setting the initialization parameter AUDIT_TRAIL to DB. You can specify auditing options regardless of whether auditing is enabled. However, Oracle does not generate audit records until you enable auditing.

Keywords and Parameters

statement_opt chooses specific SQL statements for auditing. For a list of these statement options and the SQL statements they audit, see Table 7–1 and Table 7–2.

For each audited operation, Oracle produces an audit record containing this information:

- user performing the operation
- type of operation
- object involved in the operation
- date and time of the operation

Oracle writes audit records to the audit trail, which is a database table containing audit records. You can review database activity by examining the audit trail through data dictionary views. For information on these views, see the *Oracle8i Reference*.

system_priv chooses SQL statements that are authorized by the specified system privilege for auditing. For a list of all system privileges and the SQL statements that they authorize, see Table 7–5.

Oracle provides shortcuts for specifying groups of system privileges and statement options at once. However, Oracle encourages you to choose individual system privileges and statement options for auditing, because these shortcuts may not be supported in future versions of Oracle. The shortcuts are:

CONNECT is equivalent to specifying the CREATE SESSION system privilege

RESOURCE is equivalent to specifying the following system privileges:

- ALTER SESSION
- CREATE CLUSTER
- CREATE DATABASE LINK
- CREATE PROCEDURE
- CREATE ROLLBACK SEGMENT
- CREATE SEQUENCE
- CREATE SYNONYM
- CREATE TABLE
- CREATE TABLESPACE
- CREATE VIEW

DBA is equivalent to the SYSTEM GRANT statement option and the following system privileges:

- AUDIT SYSTEM
- CREATE PUBLIC DATABASE LINK
- CREATE PUBLIC SYNONYMN
- CREATE ROLE
- CREATE USER

ALL — is equivalent to specifying all statements options shown in Table 7–1 but not the additional statement options shown in Table 7–2.

ALL PRIVILEGES — is equivalent to specifying all system privileges.

BY *user* — chooses only SQL statements issued by specified users for auditing. If you omit this clause, Oracle audits all users' statements.

BY *proxy* — chooses for auditing only SQL statements issued by the specified proxy. For more information on proxies and their use of the database, see *Oracle8i Concepts*.

ON BEHALF OF — specifies the user or users on whose behalf the proxy executes the specified statement.

- *user* specifies auditing of statements executed on behalf of a particular user.
- ANY specifies auditing of statements executed on behalf of any user.

BY SESSION — causes Oracle to write a single record for all SQL statements of the same type issued in the same session.

BY ACCESS — causes Oracle to write one record for each audited statement.

If you specify statement options or system privileges that audit data definition language (DDL) statements, Oracle automatically audits by access regardless of whether you specify the BY SESSION clause or BY ACCESS clause.

For statement options and system privileges that audit SQL statements other than DDL, you can specify either BY SESSION or BY ACCESS. BY SESSION is the default.

WHENEVER SUCCESSFUL — chooses auditing only for statements that succeed.

NOT chooses auditing only for statements that fail or result in errors.

If you omit the WHENEVER SUCCESSFUL clause, Oracle audits SQL statements regardless of success or failure.

Table 7–1 Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
CLUSTER	CREATE CLUSTER
	AUDIT CLUSTER
	DROP CLUSTER
	TRUNCATE CLUSTER
CONTEXT	CREATE CONTEXT
	DROP CONTEXT
DATABASE LINK	CREATE DATABASE LINK
	DROP DATABASE LINK
DIMENSION	CREATE DIMENSION
	ALTER DIMENSION
	DROP DIMENSION
DIRECTORY	CREATE DIRECTORY
	DROP DIRECTORY
INDEX	CREATE INDEX
	ALTER INDEX
	DROP INDEX
NOT EXISTS	All SQL statements that fail because a specified object does not exist.
PROCEDURE[a]	CREATE FUNCTION
	CREATE LIBRARY
	CREATE PACKAGE
	CREATE PACKAGE BODY
	CREATE PROCEDURE
	DROP FUNCTION
	DROP LIBRARY
	DROP PACKAGE
	DROP PROCEDURE
PROFILE	CREATE PROFILE
	ALTER PROFILE
	DROP PROFILE

Table 7–1 (Cont.) Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
PUBLIC DATABASE LINK	CREATE PUBLIC DATABASE LINK
	DROP PUBLIC DATABASE LINK
PUBLIC SYNONYM	CREATE PUBLIC SYNONYM
	DROP PUBLIC SYNONYM
ROLE	CREATE ROLE
	ALTER ROLE
	DROP ROLE
	SET ROLE
ROLLBACK STATEMENT	CREATE ROLLBACK SEGMENT
	ALTER ROLLBACK SEGMENT
	DROP ROLLBACK SEGMENT
SEQUENCE	CREATE SEQUENCE
	DROP SEQUENCE
SESSION	Logons
SYNONYM	CREATE SYNONYM
	DROP SYNONYM
SYSTEM AUDIT	AUDIT *sql_statements*
	NOAUDIT *sql_statements*
SYSTEM GRANT	GRANT *system_privileges_and_roles*
	REVOKE *system_privileges_and_roles*
TABLE	CREATE TABLE
	DROP TABLE
	TRUNCATE TABLE
	COMMENT ON TABLE
	DELETE [FROM] *table*
TABLESPACE	CREATE TABLESPACE
	ALTER TABLESPACE
	DROP TABLESPACE

Table 7–1 (Cont.) Statement Auditing Options for Database Objects

Statement Option	SQL Statements and Operations
TRIGGER	CREATE TRIGGER
	ALTER TRIGGER
	with ENABLE and DISABLE clauses
	DROP TRIGGER
	ALTER TABLE
	with ENABLE ALL TRIGGERS clause
	and DISABLE ALL TRIGGERS clause
TYPE	CREATE TYPE
	CREATE TYPE BODY
	ALTER TYPE
	DROP TYPE
	DROP TYPE BODY
USER	CREATE USER
	ALTER USER
	DROP USER
VIEW	CREATE VIEW
	DROP VIEW

[a]Java schema objects (sources, classes, and resources) are considered the same as procedures for purposes of auditing SQL statements.

Table 7–2 Additional Statement Auditing Options for SQL Statements

Statement Option	SQL Statements and Operations
ALTER SEQUENCE	ALTER SEQUENCE
ALTER TABLE	ALTER TABLE
COMMENT TABLE	COMMENT ON TABLE *table, view, snapshot*
	COMMENT ON COLUMN *table.column, view.column, snapshot.column*
DELETE TABLE	DELETE FROM *table, view*

Table 7–2 (Cont.) Additional Statement Auditing Options for SQL Statements

Statement Option	SQL Statements and Operations
EXECUTE PROCEDURE	CALL
	Execution of any procedure or function or access to any variable, library, or cursor inside a package.
GRANT DIRECTORY	GRANT privilege ON directory
	REVOKE privilege ON directory
GRANT PROCEDURE	GRANT privilege ON procedure, function, package
	REVOKE privilege ON procedure, function, package
GRANT SEQUENCE	GRANT privilege ON sequence
	REVOKE privilege ON sequence
GRANT TABLE	GRANT privilege ON table, view, snapshot.
	REVOKE privilege ON table, view, snapshot
GRANT TYPE	GRANT privilege ON TYPE
	REVOKE privilege ON TYPE
INSERT TABLE	INSERT INTO table, view
LOCK TABLE	LOCK TABLE table, view
SELECT SEQUENCE	Any statement containing sequence.CURRVAL or sequence.NEXTVAL
SELECT TABLE	SELECT FROM table, view, snapshot
UPDATE TABLE	UPDATE table, view

Examples

Role Examples To choose auditing for every SQL statement that creates, alters, drops, or sets a role, regardless of whether the statement completes successfully, issue the following statement:

```
AUDIT ROLE;
```

To choose auditing for every statement that successfully creates, alters, drops, or sets a role, issue the following statement:

```
AUDIT ROLE
    WHENEVER SUCCESSFUL;
```

To choose auditing for every CREATE ROLE, ALTER ROLE, DROP ROLE, or SET ROLE statement that results in an Oracle error, issue the following statement:

```
AUDIT ROLE
    WHENEVER NOT SUCCESSFUL;
```

Query/Update Examples To choose auditing for any statement that queries or updates any table, issue the following statement:

```
AUDIT SELECT TABLE, UPDATE TABLE;
```

To choose auditing for statements issued by the users SCOTT and BLAKE that query or update a table or view, issue the following statement:

```
AUDIT SELECT TABLE, UPDATE TABLE
    BY scott, blake;
```

Delete Example To choose auditing for statements issued using the DELETE ANY TABLE system privilege, issue the following statement:

```
AUDIT DELETE ANY TABLE;
```

Directory Examples To choose auditing for statements issued using the CREATE ANY DIRECTORY system privilege, issue the following statement:

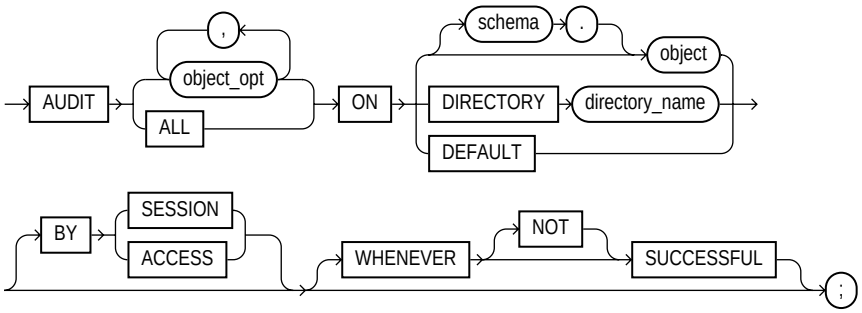
```
AUDIT CREATE ANY DIRECTORY;
```

To choose auditing for CREATE DIRECTORY (and DROP DIRECTORY) statements that do not use the CREATE ANY DIRECTORY system privilege, issue the following statement:

```
AUDIT DIRECTORY;
```

AUDIT *schema_objects*

Syntax



Purpose

To track operations on a specific schema object. To choose particular SQL statements for auditing, see "AUDIT sql_statements" on page 7-197.

Auditing keeps track of operations performed by database users. Auditing options established by the AUDIT *schema_objects* statement apply to current sessions as well as to subsequent sessions. For information on discontinuing auditing operations, see "NOAUDIT schema_objects" on page 7-525.

Prerequisites

The object you choose for auditing must be in your own schema or you must have AUDIT ANY system privilege. In addition, if the object you choose for auditing is a directory object, even if you created it, you must have AUDIT ANY system privilege.

Keywords and Parameters

object_opt specifies a particular operation for auditing. Table 7–3 shows each object option and the types of objects to which it applies. The name of each object option specifies a SQL statement to be audited. For example, if you choose to audit a table with the ALTER option, Oracle audits all ALTER TABLE statements issued against the table. If you choose to audit a sequence with the SELECT option, Oracle audits all statements that use any of the sequence's values.

ALL	is a shortcut equivalent to specifying all object options applicable for the type of object. You can use this shortcut rather than explicitly specifying all options for an object.
schema	is the schema containing the object chosen for auditing. If you omit *schema*, Oracle assumes the object is in your own schema.
object	identifies the object chosen for auditing. The object must be a table, view, sequence, stored procedure, function, package, snapshot, or library. You can also specify a synonym for a table, view, sequence, procedure, stored function, package, or snapshot.
ON DEFAULT	establishes the specified object options as default object options for subsequently created objects. Once you have established these default auditing options, any subsequently created object is automatically audited with those options. The default auditing options for a view are always the union of the auditing options for the view's base tables. If you change the default auditing options, the auditing options for previously created objects remain the same. You can change the auditing options for an existing object only by specifying the object in the ON clause of the AUDIT statement.
ON DIRECTORY *directory_name*	identifies the name of the directory chosen for auditing.
BY SESSION	causes Oracle to write a single record for all operations of the same type on the same object issued in the same session. This is the default.
BY ACCESS	causes Oracle to write one record for each audited operation.
WHENEVER SUCCESSFUL	chooses auditing only for SQL statements that complete successfully. NOT chooses auditing only for statements that fail, or result in errors. If you omit the WHENEVER SUCCESSFUL clause entirely, Oracle audits all SQL statements, regardless of success or failure.

Table 7–3 Object Auditing Options

Object Option	Table	View	Sequence	Procedure Function Package[a]	Materialized View / Snapshot	Directory	Library	Object Type	Context
ALTER	X		X		X			X	
AUDIT	X	X	X	X	X	X		X	X
COMMENT	X	X			X				
DELETE	X	X			X				
EXECUTE				X			X		
GRANT	X	X	X	X	X	X	X	X	X
INDEX	X				X				
INSERT	X	X			X				
LOCK	X	X			X				
READ						X			
RENAME	X	X		X	X				
SELECT	X	X	X		X				
UPDATE	X	X			X				

[a] Java schema objects (sources, classes, and resources) are considered the same as procedures, functions, and packages for purposes of auditing options.

Examples

Query Examples To choose auditing for every SQL statement that queries the EMP table in the schema SCOTT, issue the following statement:

```
AUDIT SELECT
    ON scott.emp;
```

To choose auditing for every statement that successfully queries the EMP table in the schema SCOTT, issue the following statement:

```
AUDIT SELECT
    ON scott.emp
    WHENEVER SUCCESSFUL;
```

To choose auditing for every statement that queries the EMP table in the schema SCOTT and results in an Oracle error, issue the following statement:

```
AUDIT SELECT
    ON scott.emp
    WHENEVER NOT SUCCESSFUL;
```

Insert/Update Example To choose auditing for every statement that inserts or updates a row in the DEPT table in the schema BLAKE, issue the following statement:

```
AUDIT INSERT, UPDATE
    ON blake.dept;
```

ALL Example To choose auditing for every statement that performs any operation on the ORDER sequence in the schema ADAMS, issue the following statement:

```
AUDIT ALL
    ON adams.order;
```

The above statement uses the ALL shortcut to choose auditing for the following statements that operate on the sequence:

- ALTER SEQUENCE
- AUDIT
- GRANT
- any statement that accesses the sequence's values using the pseudocolumns CURRVAL or NEXTVAL

READ Example To choose auditing for every statement that reads files from the BFILE_DIR1 directory, issue the following statement:

```
AUDIT READ ON DIRECTORY bfile_dir1;
```

DEFAULT Example The following statement specifies default auditing options for objects created in the future:

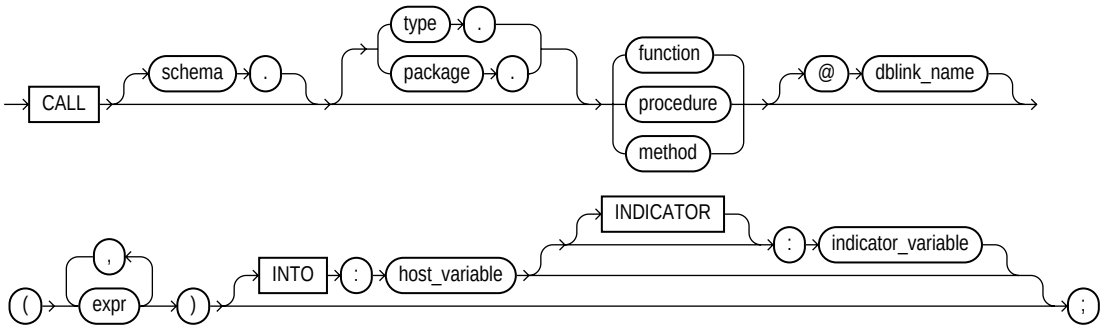
```
AUDIT ALTER, GRANT, INSERT, UPDATE, DELETE
    ON DEFAULT;
```

Any objects created later are automatically audited with the specified options that apply to them, provided that auditing has been enabled:

- If you create a table, Oracle automatically audits any ALTER, GRANT, INSERT, UPDATE, or DELETE statements issued against the table.
- If you create a view, Oracle automatically audits any GRANT, INSERT, UPDATE, or DELETE statements issued against the view.
- If you create a sequence, Oracle automatically audits any ALTER or GRANT statements issued against the sequence.
- If you create a procedure, package, or function, Oracle automatically audits any ALTER or GRANT statements issued against it.

CALL

Syntax



Purpose

Enables you to execute a **routine** (a standalone procedure or function, or a procedure or function defined within a type or package) from within SQL. For information on creating such routine, refer to *PL/SQL User's Guide and Reference.*

Prerequisites

You must have EXECUTE privilege on the standalone routine or on the type or package in which the routine is defined.

Keywords and Parameters

schema	specifies the schema in which the standalone routine (or the package or type containing the routine) resides. If you do not specify *schema*, Oracle assumes the routine is in your own schema.
type or *package*	specifies the type or package in which the routine is defined.
function | *procedure* | *method*	specifies the name of the function or procedure being called, or a synonym that translates to a function or procedure. When you call a type's member function or procedure, if the first argument (SELF) is a null IN OUT argument, Oracle returns an error. If SELF is a null IN argument, Oracle returns null. In both cases, the function or procedure is not invoked. **Restriction:** If the routine is a function, the INTO clause is mandatory.

@dblink	in a distributed database system, specifies the name of the database containing the standalone routine (or the package or functioning containing the routine). If you omit *dblink*, Oracle looks in your local database.
expr	specifies one or more arguments to the routine. **Restrictions:** ■ An *expr* cannot be a pseudocolumn or either of the correlation variables VALUE or REF. ■ Any *expr* that is an IN OUT or OUT argument of the routine must correspond to a host variable expression.
INTO :*host_variable*	applies only to calls to functions. This parameter specifies which host variable will store the return value of the function.
:*indicator_variable*	indicates the value or condition of the host variable. For more information on host variables and indicator variables, refer to *Pro*C/C++ Precompiler Programmer's Guide.*

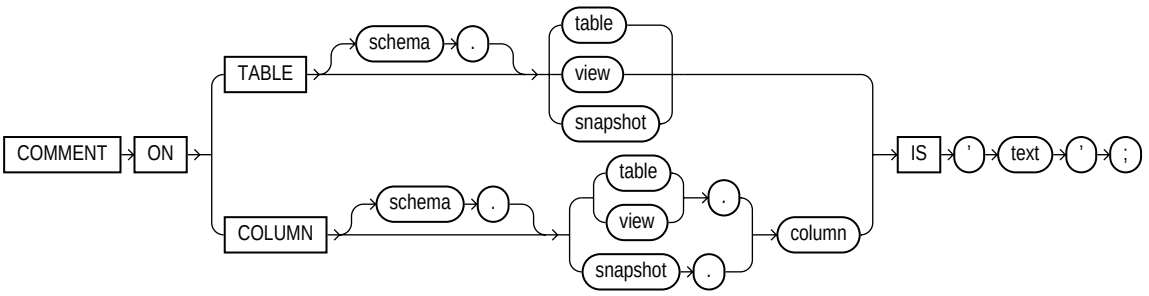
Example

The following statement creates a procedure UPDATESALARY, and then calls the procedure, which updates the specified employee ID with a new salary.

```
CREATE OR REPLACE PROCEDURE updateSalary
   (id NUMBER, newsalary NUMBER) IS
   BEGIN
      UPDATE emp SET sal=newsalary WHERE empno=id;
   END;

CALL updateSalary(1404, 50000);
```

COMMENT

Syntax



Purpose

To add a comment about a table, view, materialized view, or column into the data dictionary. See also "Comments" on page 2-56.

You can view the comments on a particular table or column by querying the data dictionary views USER_TAB_COMMENTS, DBA_TAB_COMMENTS, or ALL_TAB_COMMENTS or USER_COL_COMMENTS, DBA_COL_COMMENTS, or ALL_COL_COMMENTS. For information on these views, see *Oracle8i Reference*.

To drop a comment from the database, set it to the empty string ' '.

Prerequisites

The table, view, or snapshot must be in your own schema or you must have COMMENT ANY TABLE system privilege.

Keywords and Parameters

TABLE	specifies the schema and name of the table, view, or snapshot to be commented. If you omit *schema*, Oracle assumes the table, view, or snapshot is in your own schema.
COLUMN	specifies the name of the column of a table, view, or snapshot to be commented. If you omit *schema*, Oracle assumes the table, view, or snapshot is in your own schema.
IS '*text*'	is the text of the comment. See the syntax description of 'text' in "Text" on page 2-2.

Example

To insert an explanatory remark on the NOTES column of the SHIPPING table, you might issue the following statement:

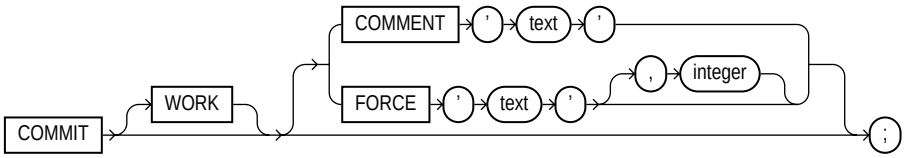
```
COMMENT ON COLUMN shipping.notes
    IS 'Special packing or shipping instructions';
```

To drop this comment from the database, issue the following statement:

```
COMMENT ON COLUMN shipping.notes IS ' ';
```

COMMIT

Syntax



Purpose

To end your current **transaction** and make permanent all changes performed in the transaction. A transaction is a sequence of SQL statements that Oracle treats as a single unit. This statement also erases all savepoints in the transaction and releases the transaction's locks. For more information on transactions, see *Oracle8i Concepts*.

> **Note:** Oracle issues an implicit COMMIT before and after any data definition language (DDL) statement.

You can also use this statement to

- Commit an in-doubt distributed transaction manually
- Terminate a read-only transaction begun by a SET TRANSACTION statement.

For more information on specifying characteristics of a transaction, see "SET TRANSACTION" on page 7-572

Oracle Corporation recommends that you explicitly end every transaction in your application programs with a COMMIT or ROLLBACK statement, including the last transaction, before disconnecting from Oracle. If you do not explicitly commit the transaction and the program terminates abnormally, the last uncommitted transaction is automatically rolled back.

A normal exit from most Oracle utilities and tools causes the current transaction to be committed. A normal exit from an Oracle precompiler program does not commit the transaction and relies on Oracle to roll back the current transaction.

Prerequisites

You need no privileges to commit your current transaction.

To manually commit a distributed in-doubt transaction that you originally committed, you must have FORCE TRANSACTION system privilege. To manually commit a distributed in-doubt transaction that was originally committed by another user, you must have FORCE ANY TRANSACTION system privilege.

Keywords and Parameters

WORK	is supported for compliance with standard SQL. The statements COMMIT and COMMIT WORK are equivalent.
COMMENT	specifies a comment to be associated with the current transaction. The 'text' is a quoted literal of up to 50 characters that Oracle stores in the data dictionary view DBA_2PC_PENDING along with the transaction ID if the transaction becomes in-doubt. For more information on adding comments to SQL statements, see "COMMENT" on page 7-212.
FORCE	in a distributed database system, manually commits an in-doubt distributed transaction. The transaction is identified by the 'text' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. You can use *integer* to specifically assign the transaction a system change number (SCN). If you omit *integer*, the transaction is committed using the current SCN. **Note:** A COMMIT statement with a FORCE clause commits only the specified transaction. Such a statement does not affect your current transaction. For more information on these topics, see *Oracle8i Distributed Database Systems*. **Restriction:** COMMIT statements using the FORCE clause are not supported in PL/SQL.

Examples

INSERT Example This statement inserts a row into the DEPT table and commits this change:

```
INSERT INTO dept VALUES (50, 'MARKETING', 'TAMPA');
COMMIT WORK;
```

COMMENT Example The following statement commits the current transaction and associates a comment with it:

```
COMMIT
    COMMENT 'In-doubt transaction Code 36, Call (415) 555-2637';
```

If a network or machine failure prevents this distributed transaction from committing properly, Oracle stores the comment in the data dictionary along with the transaction ID. The comment indicates the part of the application in which the failure occurred and provides information for contacting the administrator of the database where the transaction was committed.

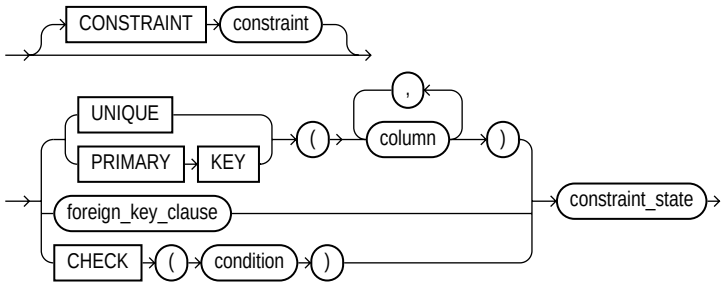
In-Doubt Transaction Example The following statement manually commits an in-doubt distributed transaction:

```
COMMIT FORCE '22.57.53';
```

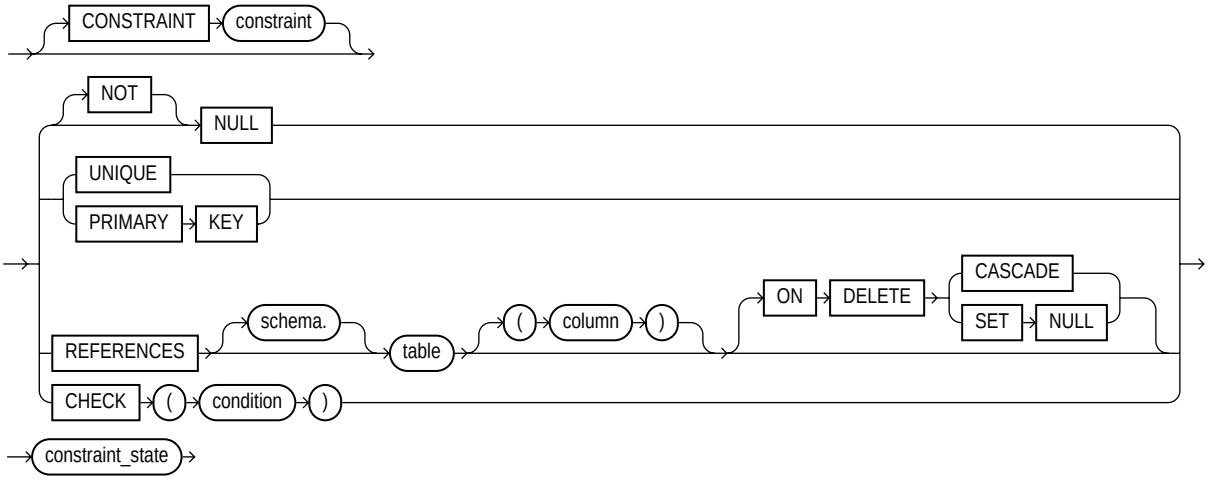
constraint_clause

Syntax

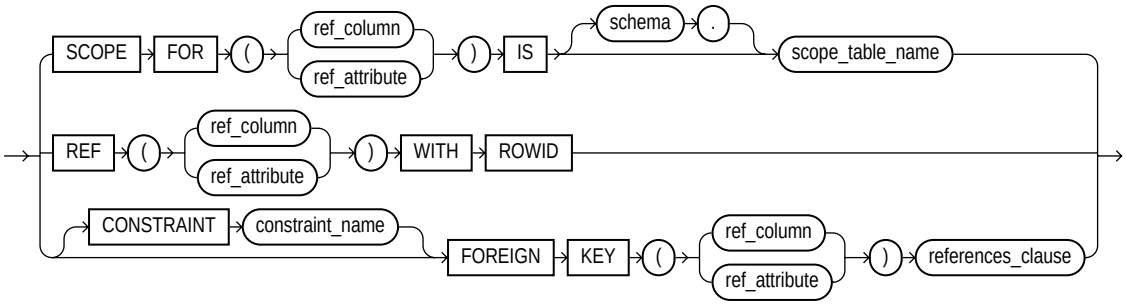
table_constraint::=



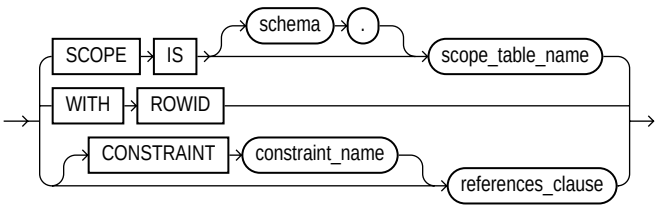
column_constraint::=



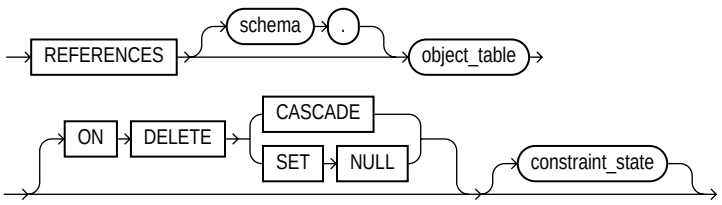
table_ref_constraint::=



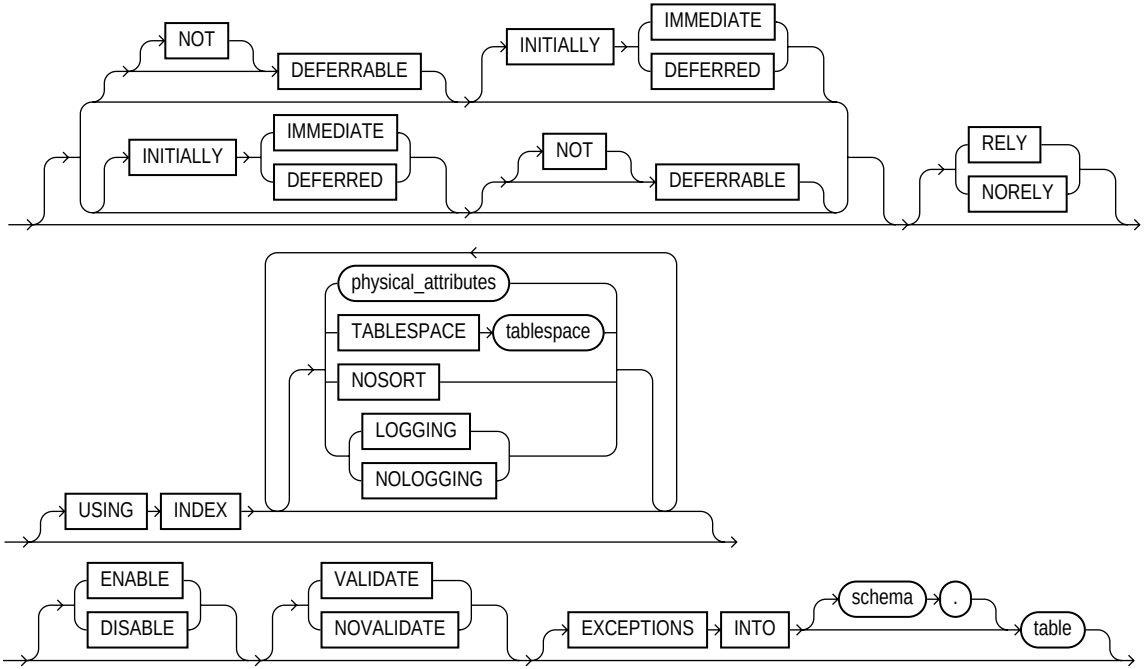
table_ref_constraint::=



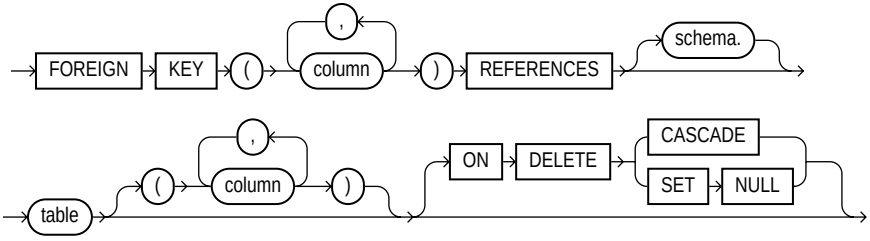
references_clause::=



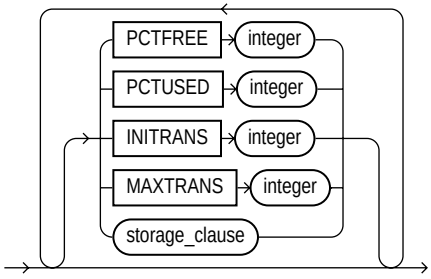
constraint_state::=



foreign_key_clause::=



physical_attributes_clause::=



storage_clause: See the "storage_clause" on page 7-575.

Purpose

To define an integrity constraint. An **integrity constraint** is a rule that restricts the values for one or more columns in a table or an index-organized table.

> **Note:** Oracle does not support constraints on columns or attributes whose type is an object, nested table, varray, REF, or LOB. The only exception is that NOT NULL constraints are supported for columns or attributes whose type is object, VARRAY, REF, or LOB.

Prerequisites

Constraint clauses can appear in either CREATE TABLE or ALTER TABLE statements. To define an integrity constraint, you must have the privileges necessary to issue one of these statements. See "CREATE TABLE" on page 7-359 and "ALTER TABLE" on page 7-113.

To create a referential integrity constraint, the parent table must be in your own schema, or you must have the REFERENCES privilege on the columns of the referenced key in the parent table.

Keywords and Parameters

table_constraint The *table_constraint* syntax is part of the table definition. An integrity constraint defined with this syntax can impose rules on any columns in the table.

The *table_constraint* syntax can appear in a CREATE TABLE or ALTER TABLE statement. This syntax can define any type of integrity constraint except a NOT NULL constraint.

column_ constraint

The *column_constraint* syntax is part of a column definition. Usually, an integrity constraint defined with this syntax can impose rules only on the column in which it is defined.

- The *column_constraint* syntax that appears in a CREATE TABLE or ALTER TABLE ADD statement can define any type of integrity constraint.
- *Column_constraint* syntax that appears in an ALTER TABLE MODIFY *column_options* statement can only define or remove a NOT NULL constraint.

CONSTRAINT

identifies the integrity constraint by the name *constraint*. Oracle stores this name in the data dictionary along with the definition of the integrity constraint. If you omit this identifier, Oracle generates a name with the form SYS_C*n*.

If you do not specify NULL or NOT NULL in a column definition, NULL is the default.

Restriction: You cannot create a constraint on columns or attributes whose type is user-defined object, LOB, or REF, with the following exceptions:

- You can specify a NOT NULL constraint on columns or attributes of user-defined object type, varray, and LOB.
- You can specify NOT NULL **and** referential integrity constraints on a column of type REF.

UNIQUE

designates a column or combination of columns as a unique key. To satisfy a UNIQUE constraint, no two rows in the table can have the same value for the unique key. However, the unique key made up of a single column can contain nulls.

A **composite unique key** is made up of a combination of columns. To define a composite unique key, you must use *table_constraint* syntax rather than *column_constraint* syntax. Any row that contains nulls in all key columns automatically satisfies the constraint. However, two rows that contain nulls for one or more key columns and the same combination of values for the other key columns violate the constraint.

Restrictions:

- For a composite unique key, no two rows in the table can have the same combination of values in the key columns.
- A composite unique key cannot have more than 32 columns. The overall size of the key (in bytes) should not exceed approximately the width of all indexed columns plus the number of indexed columns.
- A unique key column cannot be of datatype LONG or LONG RAW.
- You cannot designate the same column or combination of columns as both a unique key and a primary key.

PRIMARY KEY — designates a column or combination of columns as the table's primary key. A **composite primary key** is made up of a combination of columns. To define a composite primary key, you must use the *table_constraint* syntax rather than the *column_constraint* syntax.

Restrictions:

- A table can have only one primary key.
- None of the columns in the primary key can have datatype LONG or LONG RAW.
- No primary key value can appear in more than one row in the table.
- No column that is part of the primary key can contain a null.
- The size of the PRIMARY KEY of an index-organized table cannot exceed one-half of the database block size or 3800 bytes, whichever is less. (PRIMARY KEY is required for an index-organized table.)
- A composite primary key cannot have more than 32 columns. The overall size of the key (in bytes) should not exceed approximately the width of all indexed columns plus the number of indexed columns.
- You cannot designate the same column or combination of columns as both a primary key and a unique key.

NULL | NOT NULL — determines whether a column can contain nulls. You must specify NULL and NOT NULL with *column_constraint* syntax, not with *table_constraint* syntax.

NULL — specifies that a column can contain null values. The NULL keyword does not actually define an integrity constraint. If you do not specify either NOT NULL or NULL, the column can contain nulls by default.

NOT NULL — specifies that a column cannot contain null values. To satisfy this constraint, every row in the table must contain a value for the column.

Referential integrity constraints

Referential integrity constraints designate a column or combination of columns as the foreign key and establish a relationship between that foreign key and a specified primary or unique key, called the **referenced key**. The table containing the foreign key is called the **child table**, and the table containing the referenced key is called the **parent table**. The foreign key and the referenced key can be in the same table. In this case, the parent and child tables are the same.

From the table level, specify referential integrity using the *foreign_key_clause* with the *table_constraint* syntax. This syntax allows you to specify a **composite foreign key**, which is made up of a combination of columns.

From the column level, use the REFERENCES clause of the *column_constraint* syntax to specify a referential integrity constraint in which the foreign key is made up of a single column.

You can designate the same column or combination of columns as both a foreign key and a primary or unique key. You can also designate the same column or combination of columns as both a foreign key and a cluster key.

You can define multiple foreign keys in a table. Also, a single column can be part of more than one foreign key.

Restrictions:

- A foreign key cannot be of type LONG or LONG RAW.
- The referenced UNIQUE or PRIMARY key constraint on the parent table must already be defined.
- The child and parent tables must be on the same database. To enable referential integrity constraints across nodes of a distributed database, you must use database triggers. For more information, see *Oracle8i Application Developer's Guide - Fundamentals*.
- You cannot define a referential integrity constraint in a CREATE TABLE statement that contains an AS *subquery* clause. Instead, you must create the table without the constraint and then add it later with an ALTER TABLE statement.

foreign_key_clause designates a column or combination of columns as the foreign key from the table level. You must use this syntax to define a composite foreign key.

To satisfy a referential integrity constraint involving composite keys, either the values of the foreign key columns must match the values of the referenced key columns in a row in the parent table, or the value of at least one of the columns of the foreign key must be null.

Restrictions:

- A composite foreign key cannot have more than 32 columns. The overall size of the key (in bytes) should not exceed approximately the width of all indexed columns plus the number of indexed columns.
- A composite foreign key must refer to a composite unique key or a composite primary key.

REFERENCES designates the current column or attribute as the foreign key and identifies the parent table and the column or combination of columns that make up the referenced key. If you identify only the parent table and omit the column names, the foreign key automatically references the primary key of the parent table. The corresponding columns of the referenced key and the foreign key must match in number and datatypes.

ON DELETE — determines how Oracle automatically maintains referential integrity if you remove a referenced primary or unique key value. If you omit this clause, Oracle does not allow you to delete referenced key values in the parent table that have dependent rows in the child table.

- CASCADE specifies that Oracle removes dependent foreign key values.
- SET NULL specifies that Oracle converts dependent foreign key values to NULL.

CHECK — specifies a condition that each row in the table must satisfy. To satisfy the constraint, each row in the table must make the condition either TRUE or unknown (due to a null). For information and syntax, see "Conditions" on page 5-13. When Oracle evaluates a CHECK constraint condition for a particular row, any column names in the condition refer to the column values in that row.

If you create multiple CHECK constraints for a column, design them carefully so their purposes do not conflict. Oracle does not verify that CHECK conditions are not mutually exclusive.

Restrictions:

The condition of a CHECK constraint can refer to any column in the table, but it cannot refer to columns of other tables.

CHECK constraint conditions cannot contain the following constructs:

- Queries to refer to values in other rows
- Calls to the functions SYSDATE, UID, USER, or USERENV
- The pseudocolumns CURRVAL, NEXTVAL, LEVEL, or ROWNUM
- Date constants that are not fully specified

table_ref_constraint and *column_ref_constraint* — further describe a column of type REF. The only difference between these clauses is that you specify *table_ref_constraint* from the table level, so you must identify the REF column or attribute you are defining. You specify *column_ref_constraint* after you have already identified the REF column or attribute. Both types of constraint let you specify a SCOPE constraint, a WITH ROWID constraint, or a referential integrity constraint.

As is the case for regular table and column constraints, you use FOREIGN KEY syntax for a referential integrity constraint at the table level, and REFERENCES syntax for a referential integrity constraint at the column level. See "Referential integrity constraints" on page 7-222.

If the REF column's scope table or reference table has a primary-key-based object identifier, then it is a **user-defined REF column**. For more information on REFs, see *Oracle8i Concepts*.

ref_column — is the name of a REF column of an object or relational table.

ref_attribute — is an embedded REF attribute within an object column of a relational table.

SCOPE

In a table with a REF column, each REF value in the column can conceivably reference a row in a different object table. The SCOPE clause restricts the scope of references to a single table, *scope_table_name*. The values in the REF column or attribute point to objects in *scope_table_name*, in which object instances (of the same type as the REF column) are stored. You can only specify one scope table per REF column.

Restrictions:

- You can add a SCOPE constraint to an existing column only if the table is empty.
- You cannot specify SCOPE for the REF elements of a varray column.
- You must specify this clause if you specify AS *subquery* and the subquery returns user-defined REFs.
- The *scope_table_name* must be in your own schema or you must have SELECT privileges on *scope_table_name* or SELECT ANY TABLE system privileges.
- You cannot drop a SCOPE table constraint from a REF column.

WITH ROWID

stores the rowid along with the REF value in *ref_column* or *ref_attribute*. Storing a REF value with a rowid can improve the performance of dereferencing operations, but will also use more space. Default storage of REF values is without rowids.

Restrictions:

- You cannot specify a WITH ROWID constraint for the REF elements of a varray column.
- You cannot drop a WITH ROWID constraint from a REF column.
- If the REF column or attribute is scoped, then this clause is ignored and the rowid is not stored with the REF value.

references_clause

specifies a referential integrity constraint on the REF column.This clause also implicitly restricts the scope of the REF column or attribute to the reference table.

If you do not specify CONSTRAINT, Oracle generates a system name for the constraint.

Restrictions:

- If you add a referential integrity constraint to an existing REF column that is scoped, then the referenced table must be the same as the scope table of the REF column.
- The system adds a scope constraint when you add a referential integrity constraint to an existing unscoped REF column. Therefore, all the restrictions that apply for the SCOPE constraint also apply in this case.
- If you later drop the referential integrity constraint, the REF column will remain scoped to the referenced table.

DEFERRABLE — indicates that constraint checking can be deferred until the end of the transaction by using the SET CONSTRAINT(S) statement. For information on checking constraints after each DML statement, see "SET CONSTRAINT(S)" on page 7-568. See *Oracle8i Administrator's Guide* and *Oracle8i Concepts* for more information about deferred constraints.

NOT DEFERRABLE — indicates that this constraint is checked at the end of each DML statement. If you do not specify either word, then NOT DEFERRABLE is the default.

INITIALLY IMMEDIATE — indicates that at the start of every transaction, the default is to check this constraint at the end of every DML statement. If you do not specify INITIALLY, INITIALLY IMMEDIATE is the default.

INITIALLY DEFERRED — implies that this constraint is DEFERRABLE and specifies that, by default, the constraint is checked only at the end of each transaction.

Restrictions:

- You cannot defer a NOT DEFERRABLE constraint with the SET CONSTRAINT(S) statement.
- You cannot specify either DEFERRABLE or NOT DEFERRABLE if you are modifying an existing constraint directly (that is, by specifying the ALTER TABLE ... MODIFY *constraint* statement).
- You cannot alter a constraint's deferrability status. You must drop the constraint and re-create it.

RELY | NORELY — specifies whether an enabled constraint is to be enforced. Specify RELY to enable an existing constraint without enforcement. Specify NORELY to enable and enforce an existing constraint. The default is NORELY.

Unenforced constraints are generally useful only with materialized views and query rewrite. Depending on the QUERY_REWRITE_INTEGRITY mode (see "ALTER SESSION" on page 7-78), query rewrite can use constraints that are enabled with or without enforcement to determine join information. For more information on materialized views and query rewrite, see *Oracle8i Tuning*.

Restrictions:

- RELY and NORELY are relevant only if you are modifying an existing constraint (that is, you have issued the ALTER TABLE ... MODIFY constraint statement).
- You cannot set a NOT NULL constraint to RELY.

USING INDEX — specifies parameters for the index Oracle uses to enable a UNIQUE or PRIMARY KEY constraint. The name of the index is the same as the name of the constraint. You can choose the values of the INITRANS, MAXTRANS, TABLESPACE, STORAGE, PCTFREE, LOGGING, and NOLOGGING parameters for the index. For information on these parameters, see "CREATE TABLE" on page 7-359.

Restrictions:

- Use this clause only when enabling UNIQUE and PRIMARY KEY constraints.
- You cannot specify the PCTUSED parameter with this clause, because that parameter is not valid with indexes.

NOSORT — indicates that the rows are stored in the database in ascending order and therefore Oracle does not have to sort the rows when creating the index.

ENABLE — specifies that the constraint will be applied to all new data in the table. Before you can enable a referential integrity constraint, its referenced constraint must be enabled.

- ENABLE VALIDATE additionally specifies that all old data also complies with the constraint. An enabled validated constraint guarantees that all data is and will continue to be valid.

  If you place a primary key constraint in ENABLE VALIDATE mode, the validation process will verify that the primary key columns contain no nulls. To avoid this overhead, mark each column in the primary key NOT NULL before enabling the table's primary key constraint. (For optimal results, do this before inserting data into the column.)

- ENABLE NOVALIDATE ensures that all new DML operations on the constrained data comply with the constraint, but does not ensure that existing data in the table complies with the constraint.

Enabling a primary key or unique key constraint automatically creates a unique index to enforce the constraint. This index is dropped if the constraint is subsequently disabled, causing Oracle to rebuild the index every time the constraint is enabled. To avoid this behavior, create new primary key and unique key constraints initially disabled. Then create nonunique indexes or use existing nonunique indexes to enforce the constraints.

For additional notes and restrictions, see the *enable_disable_clause* of "CREATE TABLE" on page 7-382.

DISABLE — disables the integrity constraint. If you do not specify this clause when creating a constraint, Oracle automatically enables the constraint.

- DISABLE VALIDATE disables the constraint and drops the index on the constraint, but keeps the constraint valid. This feature is most useful in data warehousing situations, where the need arises to load into a range-partitioned table a quantity of data with a distinct range of values in the unique key. In such situations, the disable validate state enables you to save space by not having an index. You can then load data from a nonpartitioned table into a partitioned table using the *exchange_partition_clause* of the ALTER TABLE statement. All other modifications to the table by other SQL statements are disallowed.

If the unique key coincides with the partitioning key of the partitioned table, disabling the constraint saves overhead and has no detrimental effects. If the unique key does not coincide with the partitioning key, Oracle performs automatic table scans during the exchange to validate the constraint, which might offset the benefit of loading without an index.

- DISABLE NOVALIDATE signifies that Oracle makes no effort to maintain the constraint (because it is disabled) and cannot guarantee that the constraint is true (because it is not being validated). For information on when to use this setting, see *Oracle8i Tuning*.

  You cannot drop a table whose primary key is being referenced by a foreign key even if the foreign key constraint is in DISABLE NOVALIDATE state. Further, the optimizer can use constraints in DISABLE NOVALIDATE state.

- If you specify neither VALIDATE nor NOVALIDATE, the default is NOVALIDATE.
- If you disable a unique or primary key constraint that is using a unique index, Oracle drops the unique index.

EXCEPTIONS INTO — specifies a table into which Oracle places the ROWIDs of all rows violating the constraint.

Note: You must create an appropriate exceptions report table to accept information from the EXCEPTIONS INTO clause before enabling the constraint. You can create an exception table by submitting the script UTLEXCPT1.SQL, which creates a table named EXCEPTIONS. You can create additional exceptions tables with different names by modifying and resubmitting the script. (You can use the UTLEXCPT1.SQL script with index-organized tables. You could not use earlier versions of the script for this purpose. See *Oracle8i Migration* for compatibility information.)

This clause is valid only when validating a constraint.

Examples

NOT NULL Example The following statement alters the EMP table and defines and enables a NOT NULL constraint on the SAL column:

```
ALTER TABLE emp
   MODIFY (sal  NUMBER  CONSTRAINT nn_sal NOT NULL);
```

NN_SAL ensures that no employee in the table has a null salary.

Unique Key Example The following statement creates the DEPT table and defines and enables a unique key on the DNAME column:

```
CREATE TABLE dept
    (deptno  NUMBER(2),
     dname   VARCHAR2(9)  CONSTRAINT unq_dname UNIQUE,
```

```
      loc     VARCHAR2(10) );
```

The constraint UNQ_DNAME identifies the DNAME column as a unique key. This constraint ensures that no two departments in the table have the same name. However, the constraint does allow departments without names.

Alternatively, you can define and enable this constraint with the *table_constraint* syntax:

```
CREATE TABLE dept
    (deptno  NUMBER(2),
     dname   VARCHAR2(9),
     loc     VARCHAR2(10),
        CONSTRAINT unq_dname
        UNIQUE (dname)
    USING INDEX PCTFREE 20
        TABLESPACE user_x
        STORAGE (INITIAL 8K  NEXT 6K) );
```

The above statement also uses the USING INDEX clause to specify storage characteristics for the index that Oracle creates to enable the constraint.

Composite Unique Key Example The following statement defines and enables a composite unique key on the combination of the CITY and STATE columns of the CENSUS table:

```
ALTER TABLE census
    ADD CONSTRAINT unq_city_state
    UNIQUE (city, state)
    USING INDEX PCTFREE 5
        TABLESPACE user_y
    EXCEPTIONS INTO bad_keys_in_ship_cont;
```

The UNQ_CITY_STATE constraint ensures that the same combination of CITY and STATE values does not appear in the table more than once.

The ADD CONSTRAINT clause also specifies other properties of the constraint:

- The USING INDEX clause specifies storage characteristics for the index Oracle creates to enable the constraint.
- The EXCEPTIONS INTO clause causes Oracle to write information to the BAD_KEYS_IN_SHIP_CONT table about any rows currently in the CENSUS table that violate the constraint.

Primary Key Example The following statement creates the DEPT table and defines and enables a primary key on the DEPTNO column:

```
CREATE TABLE dept
    (deptno  NUMBER(2) CONSTRAINT pk_dept PRIMARY KEY,
     dname   VARCHAR2(9),
     loc     VARCHAR2(10) );
```

The PK_DEPT constraint identifies the DEPTNO column as the primary key of the DEPT table. This constraint ensures that no two departments in the table have the same department number and that no department number is NULL.

Alternatively, you can define and enable this constraint with *table_constraint* syntax:

```
CREATE TABLE dept
    (deptno  NUMBER(2),
    dname   VARCHAR2(9),
    loc   VARCHAR2(10),
    CONSTRAINT pk_dept PRIMARY KEY (deptno) );
```

Composite Primary Key Example The following statement defines a composite primary key on the combination of the SHIP_NO and CONTAINER_NO columns of the SHIP_CONT table:

```
ALTER TABLE ship_cont
    ADD PRIMARY KEY (ship_no, container_no) DISABLE;
```

This constraint identifies the combination of the SHIP_NO and CONTAINER_NO columns as the primary key of the SHIP_CONT table. The constraint ensures that no two rows in the table have the same values for both the SHIP_NO column and the CONTAINER_NO column.

The CONSTRAINT clause also specifies the following properties of the constraint:

- The constraint definition does not include a constraint name, so Oracle generates a name for the constraint.
- The DISABLE clause causes Oracle to define the constraint but not enable it.

Referential Integrity Constraint Example The following statement creates the EMP table and defines and enables a foreign key on the DEPTNO column that references the primary key on the DEPTNO column of the DEPT table:

```
CREATE TABLE emp
   (empno      NUMBER(4),
    ename      VARCHAR2(10),
    job        VARCHAR2(9),
```

```
    mgr        NUMBER(4),
    hiredate   DATE,
    sal        NUMBER(7,2),
    comm       NUMBER(7,2),
    deptno     CONSTRAINT fk_deptno REFERENCES dept(deptno) );
```

The constraint FK_DEPTNO ensures that all departments given for employees in the EMP table are present in the DEPT table. However, employees can have null department numbers, meaning they are not assigned to any department. To ensure that all employees are assigned to a department, you could create a NOT NULL constraint on the DEPTNO column in the EMP table, in addition to the REFERENCES constraint.

Before you define and enable this constraint, you must define and enable a constraint that designates the DEPTNO column of the DEPT table as a primary or unique key.

The referential integrity constraint definition does not use the FOREIGN KEY keyword to identify the columns that make up the foreign key. Because the constraint is defined with a column constraint clause on the DEPTNO column, the foreign key is automatically on the DEPTNO column.

The constraint definition identifies both the parent table and the columns of the referenced key. Because the referenced key is the parent table's primary key, the referenced key column names are optional.

The above statement omits the DEPTNO column's datatype. Because this column is a foreign key, Oracle automatically assigns it the datatype of the DEPT.DEPTNO column to which the foreign key refers.

Alternatively, you can define a referential integrity constraint with *table_constraint* syntax:

```
CREATE TABLE emp
  (empno     NUMBER(4),
   ename     VARCHAR2(10),
   job       VARCHAR2(9),
   mgr       NUMBER(4),
   hiredate  DATE,
   sal       NUMBER(7,2),
   comm      NUMBER(7,2),
   deptno,
   CONSTRAINT fk_deptno
      FOREIGN  KEY (deptno)
      REFERENCES  dept(deptno) );
```

The foreign key definitions in both statements of this statement omit the ON DELETE clause, causing Oracle to forbid the deletion of a department if any employee works in that department.

ON DELETE Example This statement creates the EMP table, defines and enables two referential integrity constraints, and uses the ON DELETE clause:

```
CREATE TABLE emp
  (empno    NUMBER(4) PRIMARY KEY,
   ename    VARCHAR2(10),
   job      VARCHAR2(9),
   mgr      NUMBER(4) CONSTRAINT fk_mgr
            REFERENCES emp ON DELETE SET NULL,
   hiredate DATE,
   sal      NUMBER(7,2),
   comm     NUMBER(7,2),
   deptno   NUMBER(2)   CONSTRAINT fk_deptno
            REFERENCES dept(deptno)
            ON DELETE CASCADE );
```

Because of the first ON DELETE clause, if manager number 2332 is deleted from the EMP table, Oracle sets to null the value of MGR for all employees in the EMP table who previously had manager 2332.

Because of the second ON DELETE clause, Oracle cascades any deletion of a DEPTNO value in the DEPT table to the DEPTNO values of its dependent rows of the EMP table. For example, if Department 20 is deleted from the DEPT table, Oracle deletes the department's employees from the EMP table.

Composite Referential Integrity Constraint Example The following statement defines and enables a foreign key on the combination of the AREACO and PHONENO columns of the PHONE_CALLS table:

```
ALTER TABLE phone_calls
    ADD CONSTRAINT fk_areaco_phoneno
        FOREIGN KEY (areaco, phoneno)
        REFERENCES customers(areaco, phoneno)
        EXCEPTIONS INTO wrong_numbers;
```

The constraint FK_AREACO_PHONENO ensures that all the calls in the PHONE_CALLS table are made from phone numbers that are listed in the CUSTOMERS table. Before you define and enable this constraint, you must define and enable a constraint that designates the combination of the AREACO and PHONENO columns of the CUSTOMERS table as a primary or unique key.

The EXCEPTIONS INTO clause causes Oracle to write information to the WRONG_NUMBERS table about any rows in the PHONE_CALLS table that violate the constraint.

CHECK Constraint Examples The following statement creates the DEPT table and defines a CHECK constraint in each of the table's columns:

```
CREATE TABLE dept
   (deptno NUMBER  CONSTRAINT check_deptno
          CHECK (deptno BETWEEN 10 AND 99)
          DISABLE,
    dname VARCHAR2(9)  CONSTRAINT check_dname
          CHECK (dname = UPPER(dname))
          DISABLE,
    loc VARCHAR2(10)  CONSTRAINT check_loc
          CHECK (loc IN ('DALLAS','BOSTON',
          'NEW YORK','CHICAGO'))
          DISABLE);
```

Each constraint restricts the values of the column in which it is defined:

- CHECK_DEPTNO ensures that no department numbers are less than 10 or greater than 99.
- CHECK_DNAME ensures that all department names are in uppercase.
- CHECK_LOC restricts department locations to Dallas, Boston, New York, or Chicago.

Because each CONSTRAINT clause contains the DISABLE clause, Oracle only defines the constraints and does not enable them.

The following statement creates the EMP table and uses a *table_constraint_clause* to define and enable a CHECK constraint:

```
CREATE TABLE emp
    (empno          NUMBER(4),
     ename          VARCHAR2(10),
     job            VARCHAR2(9),
     mgr            NUMBER(4),
     hiredate       DATE,
     sal            NUMBER(7,2),
     comm           NUMBER(7,2),
     deptno         NUMBER(2),
     CHECK (sal + comm <= 5000) );
```

This constraint uses an inequality condition to limit an employee's total compensation, the sum of salary and commission, to $5000:

- If an employee has non-null values for both salary and commission, the sum of these values must not exceed $5000 to satisfy the constraint.
- If an employee has a null salary or commission, the result of the condition is unknown and the employee automatically satisfies the constraint.

Because the CONSTRAINT clause in this example does not supply a constraint name, Oracle generates a name for the constraint.

The following statement defines and enables a PRIMARY KEY constraint, two referential integrity constraints, a NOT NULL constraint, and two CHECK constraints:

```
CREATE TABLE order_detail
  (CONSTRAINT pk_od PRIMARY KEY (order_id, part_no),
   order_id NUMBER
      CONSTRAINT fk_oid REFERENCES scott.order (order_id),
   part_no            NUMBER
      CONSTRAINT fk_pno REFERENCES scott.part (part_no),
   quantity            NUMBER
      CONSTRAINT nn_qty NOT NULL
      CONSTRAINT check_qty_low CHECK (quantity > 0),
   cost            NUMBER
      CONSTRAINT check_cost CHECK (cost > 0) );
```

The constraints enable the following rules on table data:

- PK_OD identifies the combination of the ORDER_ID and PART_NO columns as the primary key of the table. To satisfy this constraint, no two rows in the table can contain the same combination of values in the ORDER_ID and the PART_NO columns, and no row in the table can have a null in either the ORDER_ID column or the PART_NO column.
- FK_OID identifies the ORDER_ID column as a foreign key that references the ORDER_ID column in the ORDER table in SCOTT's schema. All new values added to the column ORDER_DETAIL.ORDER_ID must already appear in the column SCOTT.ORDER.ORDER_ID.
- FK_PNO identifies the PART_NO column as a foreign key that references the PART_NO column in the PART table owned by SCOTT. All new values added to the column ORDER_DETAIL.PART_NO must already appear in the column SCOTT.PART.PART_NO.
- NN_QTY forbids nulls in the QUANTITY column.

- CHECK_QTY ensures that values in the QUANTITY column are always greater than zero.
- CHECK_COST ensures the values in the COST column are always greater than zero.

This example also illustrates the following points about constraint clauses and column definitions:

- *Table_constraint* syntax and column definitions can appear in any order. In this example, the *table_constraint* syntax that defines the PK_OD constraint precedes the column definitions.
- A column definition can use *column_constraint* syntax multiple times. In this example, the definition of the QUANTITY column contains the definitions of both the NN_QTY and CHECK_QTY constraints.
- A table can have multiple CHECK constraints. Multiple CHECK constraints, each with a simple condition enforcing a single business rule, is better than a single CHECK constraint with a complicated condition enforcing multiple business rules. When a constraint is violated, Oracle returns an error identifying the constraint. Such an error more precisely identifies the violated business rule if the identified constraint enables a single business rule.

DEFERRABLE Constraint Examples The following statement creates table GAMES with a NOT DEFERRABLE INITIALLY IMMEDIATE constraint check on the SCORES column:

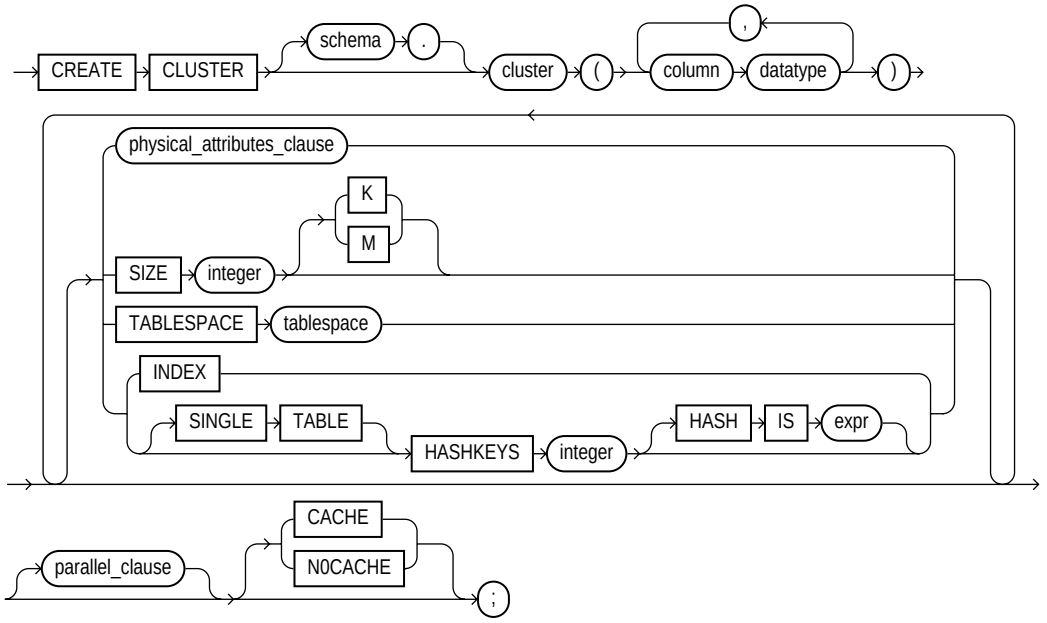
```
CREATE TABLE games (scores NUMBER CHECK (scores >= 0));
```

To define a unique constraint on a column as INITIALLY DEFERRED DEFERRABLE, issue the following statement:

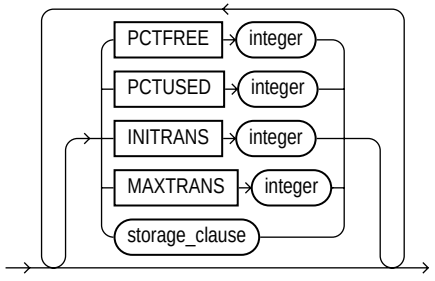
```
CREATE TABLE orders
  (ord_num NUMBER CONSTRAINT unq_num UNIQUE (ord_num)
   INITIALLY DEFERRED DEFERRABLE);
```

CREATE CLUSTER

Syntax

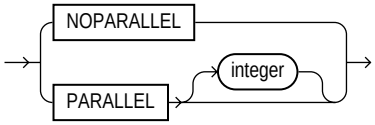


physical_attributes_clause::=



storage_clause: See the "storage_clause" on page 7-575.

parallel_clause::=



Purpose

To create a cluster. A **cluster** is a schema object that contains data from one or more tables, all of which have one or more columns in common. Oracle stores together all the rows (from all the tables) that share the same cluster key.

For general information on clusters, see *Oracle8i Concepts*. For information on performance considerations of clusters, see *Oracle8i Application Developer's Guide - Fundamentals*. For suggestions on when to use clusters, see *Oracle8i Tuning*.

Prerequisites

To create a cluster in your own schema, you must have CREATE CLUSTER system privilege. To create a cluster in another user's schema, you must have CREATE ANY CLUSTER system privilege. Also, the owner of the schema to contain the cluster must have either space quota on the tablespace containing the cluster or UNLIMITED TABLESPACE system privilege.

Oracle does not automatically create an index for a cluster when the cluster is initially created. Data manipulation language (DML) statements cannot be issued against clustered tables until a cluster index has been created.

Keywords and Parameters

schema is the schema to contain the cluster. If you omit *schema*, Oracle creates the cluster in your current schema.

cluster is the name of the cluster to be created.

After you create a cluster, you add tables to it. A cluster can contain a maximum of 32 tables. After you create a cluster and add tables to it, the cluster is transparent. You can access clustered tables with SQL statements just as you can nonclustered tables. For information on adding tables to a cluster, see "CREATE TABLE" on page 7-359.

column	is the name of a column in the cluster key. You can specify up to 16 cluster key columns. These columns must correspond in both datatype and size to columns in each of the clustered tables, although they need not correspond in name. You cannot specify integrity constraints as part of the definition of a cluster key column. Instead, you can associate integrity constraints with the tables that belong to the cluster.
datatype	is the datatype of a cluster key column. For information on datatypes, see the section "Datatypes" on page 2-5. **Restrictions:** ■ You cannot specify a cluster key column of datatype LONG, LONG RAW, REF, nested table, varray, BLOB, CLOB, BFILE, or user-defined object type. ■ You cannot use the HASH IS clause if any column datatype is not INTEGER or NUMBER with scale 0. ■ You can specify a column of type ROWID, but Oracle does not guarantee that the values in such columns are valid rowids.
physical_ attributes_clause	specifies the storage characteristics of the cluster. Each table in the cluster uses these storage characteristics as well.
PCTUSED	specifies the limit that Oracle uses to determine when additional rows can be added to a cluster's data block. The value of this parameter is expressed as a whole number and interpreted as a percentage.
PCTFREE	specifies the space reserved in each of the cluster's data blocks for future expansion. The value of the parameter is expressed as a whole number and interpreted as a percentage.
INITRANS	specifies the initial number of concurrent update transactions allocated for data blocks of the cluster. The value of this parameter for a cluster cannot be less than 2 or more than the value of the MAXTRANS parameter. The default value is 2 or the INITRANS value for the cluster's tablespace, whichever is greater.
MAXTRANS	specifies the maximum number of concurrent update transactions for any given data block belonging to the cluster. The value of this parameter cannot be less than the value of the INITRANS parameter. The maximum value of this parameter is 255. The default value is the MAXTRANS value for the tablespace to contain the cluster. For a complete description of the PCTUSED, PCTFREE, INITRANS, and MAXTRANS parameters, see "CREATE TABLE" on page 7-359.
storage_clause	specifies how data blocks are allocated to the cluster. See the "storage_clause" on page 7-575.
SIZE	specifies the amount of space in bytes to store all rows with the same cluster key value or the same hash value. Use K or M to specify this space in kilobytes or megabytes. This space determines the maximum number of cluster or hash values stored in a data block. If SIZE is not a divisor of the data block size, Oracle uses the next largest divisor. If SIZE is larger than the data block size, Oracle uses the operating system block size, reserving at least one data block per cluster or hash value.

Oracle also considers the length of the cluster key when determining how much space to reserve for the rows having a cluster key value. Larger cluster keys require larger sizes. To see the actual size, query the KEY_SIZE column of the USER_CLUSTERS data dictionary view. (This does not apply to hash clusters, because hash values are not actually stored in the cluster.)

If you omit this parameter, Oracle reserves one data block for each cluster key value or hash value.

TABLESPACE specifies the tablespace in which the cluster is created.

INDEX creates an **indexed cluster**. In an indexed cluster, Oracle stores together rows having the same cluster key value. Each distinct cluster key value is stored only once in each data block, regardless of the number of tables and rows in which it occurs.

After you create an indexed cluster, you must create an index on the cluster key before you can issue any data manipulation language (DML) statements against a table in the cluster. This index is called the *cluster index*. For information on creating a cluster index, see "CREATE INDEX" on page 7-273.

Note: You cannot create a cluster index for a hash cluster, and you need not create an index on a hash cluster key. If you specify neither INDEX nor HASHKEYS, Oracle creates an indexed cluster by default.

For more information in indexed clusters, see *Oracle8i Concepts*.

HASHKEYS creates a **hash cluster** and specifies the number of hash values for a hash cluster. In a hash cluster, Oracle stores together rows that have the same hash key value. The hash value for a row is the value returned by the cluster's hash function.

Oracle rounds up the HASHKEYS value to the nearest prime number to obtain the actual number of hash values. The minimum value for this parameter is 2. If you omit both the INDEX clause and the HASHKEYS parameter, Oracle creates an indexed cluster by default.

When you create a hash cluster, Oracle immediately allocates space for the cluster based on the values of the SIZE and HASHKEYS parameters. For more information on how Oracle allocates space for clusters, see *Oracle8i Concepts*.

SINGLE TABLE specifies that the cluster is a type of hash cluster containing only one table. This clause can provide faster access to rows than would result if the table were not part of a cluster.

Restriction: Only one table can be present in the cluster at a time. However, you can drop the table and create a different table in the same cluster.

HASH IS specifies an expression to be used as the hash function for the hash cluster. The expression:

- Must evaluate to a positive value
- Must contain at least one column with referenced columns of any datatype as long as the entire expression evaluates to a number of scale 0. For example: NUM_COLUMN * length(VARCHAR2_ COLUMN)
- Cannot reference user-defined PL/SQL functions
- Cannot reference SYSDATE, USERENV, TO_DATE, UID, USER, LEVEL, or ROWNUM
- Cannot evaluate to a constant
- Cannot contain a subquery
- Cannot contain columns qualified with a schema or object name (other than the cluster name)

If you omit the HASH IS clause, Oracle uses an internal hash function for the hash cluster.

The cluster key of a hash column can have one or more columns of any datatype. Hash clusters with composite cluster keys or cluster keys made up of noninteger columns must use the internal hash function.

parallel_clause causes creation of the cluster to be parallelized. See also the Notes to the *parallel_clause* of "CREATE TABLE" on page 7-359.

NOPARALLEL specifies serial execution. This is the default.

PARALLEL causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* specifies the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Restriction: If the tables in *cluster* contain any columns of LOB or user-defined object type, this statement as well as subsequent INSERT, UPDATE, or DELETE operations on *cluster* are executed serially without notification.

CACHE specifies that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This clause is useful for small lookup tables.

NOCACHE specifies that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the default behavior.

Examples

Creating a Cluster The following statement creates an indexed cluster named PERSONNEL with the cluster key column DEPARTMENT_NUMBER, a cluster size of 512 bytes, and storage parameter values:

```
CREATE CLUSTER personnel
    ( department_number  NUMBER(2) )
    SIZE 512
    STORAGE (INITIAL 100K NEXT 50K);
```

Adding Tables to a Cluster The following statements add the EMP and DEPT tables to the cluster:

```
CREATE TABLE emp
    (empno     NUMBER        PRIMARY KEY,
     ename     VARCHAR2(10)  NOT NULL
                             CHECK (ename = UPPER(ename)),
     job       VARCHAR2(9),
     mgr       NUMBER        REFERENCES scott.emp(empno),
     hiredate  DATE
        CHECK (hiredate < TO_DATE ('08-14-1998', 'MM-DD-YYYY')),
     sal       NUMBER(10,2)  CHECK (sal > 500),
     comm      NUMBER(9,0)   DEFAULT NULL,
     deptno    NUMBER(2)      NOT NULL )
     CLUSTER personnel (deptno);

CREATE TABLE dept
    (deptno  NUMBER(2),
     dname   VARCHAR2(9),
     loc     VARCHAR2(9))
     CLUSTER personnel (deptno);
```

Cluster Key Example The following statement creates the cluster index on the cluster key of PERSONNEL:

```
CREATE INDEX idx_personnel ON CLUSTER personnel;
```

After creating the cluster index, you can insert rows into either the EMP or DEPT tables.

Hash Cluster Examples The following statement creates a hash cluster named PERSONNEL with the cluster key column DEPARTMENT_NUMBER, a maximum

of 503 hash key values, each of which is allocated 512 bytes, and storage parameter values:

```
CREATE CLUSTER personnel
( department_number  NUMBER )
    SIZE 512  HASHKEYS 500
    STORAGE (INITIAL 100K  NEXT 50K);
```

Because the above statement omits the HASH IS clause, Oracle uses the internal hash function for the cluster.

The following statement creates a hash cluster named PERSONNEL with the cluster key made up of the columns HOME_AREA_CODE and HOME_PREFIX, and uses a SQL expression containing these columns for the hash function:

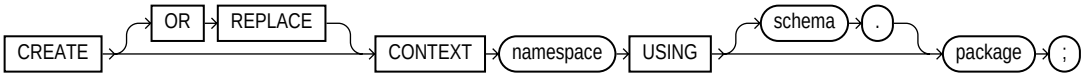
```
CREATE CLUSTER personnel
    ( home_area_code  NUMBER,
      home_prefix     NUMBER )
    HASHKEYS 20
    HASH IS MOD(home_area_code + home_prefix, 101);
```

Single-Table Hash Cluster Example The following statement creates a single-table hash cluster named PERSONNEL with the cluster key DEPTNO and a maximum of 503 hash key values, each of which is allocated 512 bytes:

```
CREATE CLUSTER personnel
   (deptno NUMBER)
   SIZE 512 SINGLE TABLE HASHKEYS 500;
```

CREATE CONTEXT

Syntax



Purpose

To create a namespace for a **context** (a set of application-defined attributes that validates and secures an application) and to associate the namespace with the externally created package that sets the context. For a definition and discussion of contexts, refer to *Oracle8i Concepts*.

Prerequisites

To create a context namespace, you must have CREATE ANY CONTEXT system privilege.

Keywords and Parameters

OR REPLACE	redefines an existing context namespace using a different package.
namespace	is the name of the context namespace to create or modify. Context namespaces are always stored in the schema SYS.
schema	is the schema owning *package*. If you omit *schema*, Oracle uses the current schema.
package	is the PL/SQL package that sets or resets the context attributes under the namespace for a user session. For more information on setting the package, see *Oracle8i Supplied Packages Reference*.

Note: To provide some design flexibility, Oracle does not verify the existence of the schema or the validity of the package at the time you create the context.

Examples

Suppose you have a human resources application (HR) and a PL/SQL package (HR_SECURE_CONTEXT), which validates and secures the HR application. The following statement creates the context namespace HR_CONTEXT and associates it with the package HR_SECURE_CONTEXT:

```
CREATE CONTEXT hr_context USING hr_secure_context;
```

You can control data access based on this context using the SYS_CONTEXT function. For example, suppose your HR_SECURE_CONTEXT package has defined an attribute ORG_ID as a particular organization identifier. You can secure a base table HR_ORG_UNIT by creating a view that restricts access based on the value of ORG_ID, as follows:

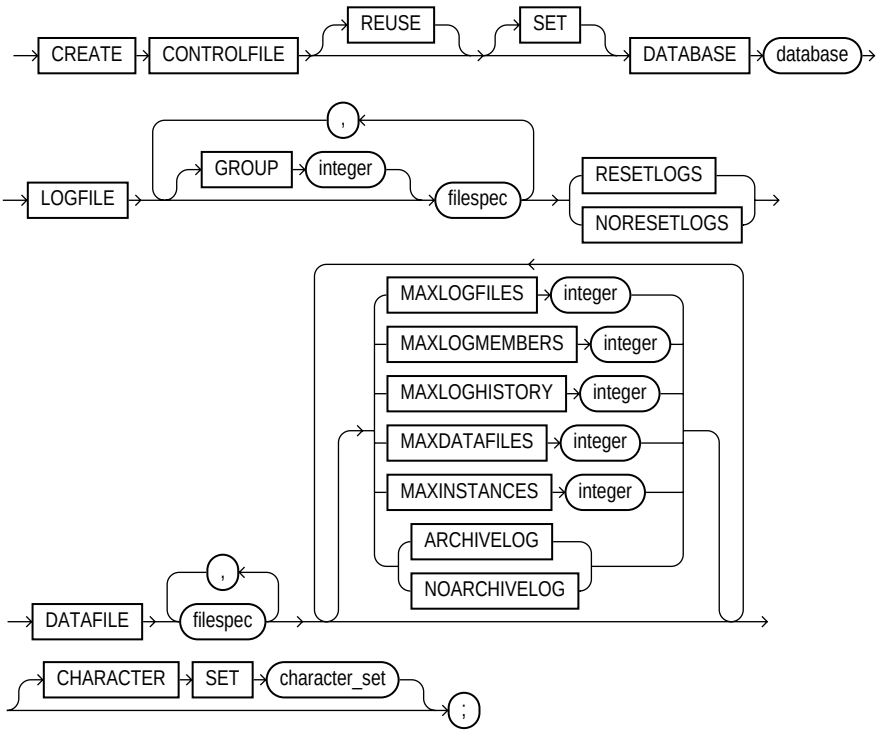
```
CREATE VIEW hr_org_secure_view AS
     SELECT * FROM hr_org_unit
     WHERE organization_id = SYS_CONTEXT ('hr_context', 'org_id');
```

For more information on the SYS_CONTEXT function, see "SYS_CONTEXT" on page 4-40.

CREATE CONTROLFILE

Syntax

WARNING: Oracle recommends that you perform a full backup of all files in the database before using this statement. For more information, see *Oracle8i Backup and Recovery Guide***.**



filespec: See "filespec" on page 7-490.

Purpose

To re-create a control file in one of the following cases:

- All copies of your existing control files have been lost through media failure.
- You want to change the name of the database.
- You want to change the maximum number of redo log file groups, redo log file members, archived redo log files, datafiles, or instances that can concurrently have the database mounted and open.

When you issue a CREATE CONTROLFILE statement, Oracle creates a new control file based on the information you specify in the statement. If you omit any clauses, Oracle uses the default values rather than the values for the previous control file. After successfully creating the control file, Oracle mounts the database in the mode specified by the initialization parameter PARALLEL_SERVER. You then must perform media recovery before opening the database. It is recommended that you then shut down the instance and take a full backup of all files in the database.

For more information about using this statement, see *Oracle8i Backup and Recovery Guide*.

Prerequisites

You must have the OSDBA role enabled. The database must not be mounted by any instance.

If the REMOTE_LOGIN_PASSWORDFILE initialization parameter is set to EXCLUSIVE, Oracle returns an error when you attempt to re-create the control file. To avoid this message, either set the parameter to SHARED, or re-create your password file before re-creating the control file. For more information about the REMOTE_LOGIN_PASSWORDFILE parameter, see *Oracle8i Reference*.

Keywords and Parameters

REUSE	specifies that existing control files identified by the initialization parameter CONTROL_FILES can be reused, thus ignoring and overwriting any information they may currently contain. If you omit this clause and any of these control files already exists, Oracle returns an error.
DATABASE *database*	specifies the name of the database. The value of this parameter must be the existing database name established by the previous CREATE DATABASE statement or CREATE CONTROLFILE statement.
SET DATABASE *database*	changes the name of the database. The name of a database can be as long as eight bytes.
LOGFILE	specifies the redo log files for your database. You must list all members of all redo log file groups. See the syntax description of filespec in "filespec" on page 7-490.

	GROUP *integer*	specifies logfile group. If you specify GROUP values, Oracle verifies these values with the GROUP values when the database was last open.
RESETLOGS	ignores the contents of the files listed in the LOGFILE clause. These files do not have to exist. Each filespec in the LOGFILE clause must specify the SIZE parameter. Oracle assigns all online redo log file groups to thread 1 and enables this thread for public use by any instance. After using this clause, you must open the database using the RESETLOGS clause of the ALTER DATABASE statement.	
NORESETLOGS	specifies that all files in the LOGFILE clause should be used as they were when the database was last open. These files must exist and must be the current online redo log files rather than restored backups. Oracle reassigns the redo log file groups to the threads to which they were previously assigned and reenables the threads as they were previously enabled.	
DATAFILE	specifies the datafiles of the database. You must list all datafiles. These files must all exist, although they may be restored backups that require media recovery. See the syntax description of *filespec* in "filespec" on page 7-490.	
MAXLOGFILES *integer*	specifies the maximum number of online redo log file groups that can ever be created for the database. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The default and maximum values depend on your operating system. The value that you specify should not be less than the greatest GROUP value for any redo log file group.	
MAXLOGMEMBERS *integer*	specifies the maximum number of members, or identical copies, for a redo log file group. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.	
MAXLOGHISTORY *integer*	specifies the maximum number of archived redo log file groups for automatic media recovery of the Oracle Parallel Server. Oracle uses this value to determine how much space in the control file to allocate for the names of archived redo log files. The minimum value is 0. The default value is a multiple of the MAXINSTANCES value and depends on your operating system. The maximum value is limited only by the maximum size of the control file. This parameter is useful only if you are using Oracle with the Parallel Server option in both parallel mode and archivelog mode.	
MAXDATAFILES *integer*	specifies the initial sizing of the datafiles section of the control file at CREATE DATABASE or CREATE CONTROLFILE time. An attempt to add a file whose number is greater than MAXDATAFILES, but less than or equal to DB_FILES, causes the Oracle control file to expand automatically so that the datafiles section can accommodate more files. The number of datafiles accessible to your instance is also limited by the initialization parameter DB_FILES.	

MAXINSTANCES *integer*	specifies the maximum number of instances that can simultaneously have the database mounted and open. This value takes precedence over the value of the initialization parameter INSTANCES. The minimum value is 1. The maximum and default values depend on your operating system.
ARCHIVELOG	establishes the mode of archiving the contents of redo log files before reusing them. This clause prepares for the possibility of media recovery as well as instance or crash recovery.
NOARCHIVELOG	If you omit both the ARCHIVELOG clause and NOARCHIVELOG clause, Oracle chooses NOARCHIVELOG mode by default. After creating the control file, you can change between ARCHIVELOG mode and NOARCHIVELOG mode with the ALTER DATABASE statement.
CHARACTER SET *character_set*	optionally reconstructs character set information in the control file. In case media recovery of the database is required, this information will be available before the database is open, so that tablespace names can be correctly interpreted during recovery. This clause is useful only if you are using a character set other than the default US7ASCII. If you are re-creating your control file and you are using Recovery Manager for tablespace recovery, and if you specify a different character set from the one stored in the data dictionary, then tablespace recovery will not succeed. (However, at database open, the control file character set will be updated with the correct character set from the data dictionary.) For more information on tablespace recovery, see *Oracle8i Backup and Recovery Guide* **Note:** You cannot modify the character set of the database with this clause.

Example

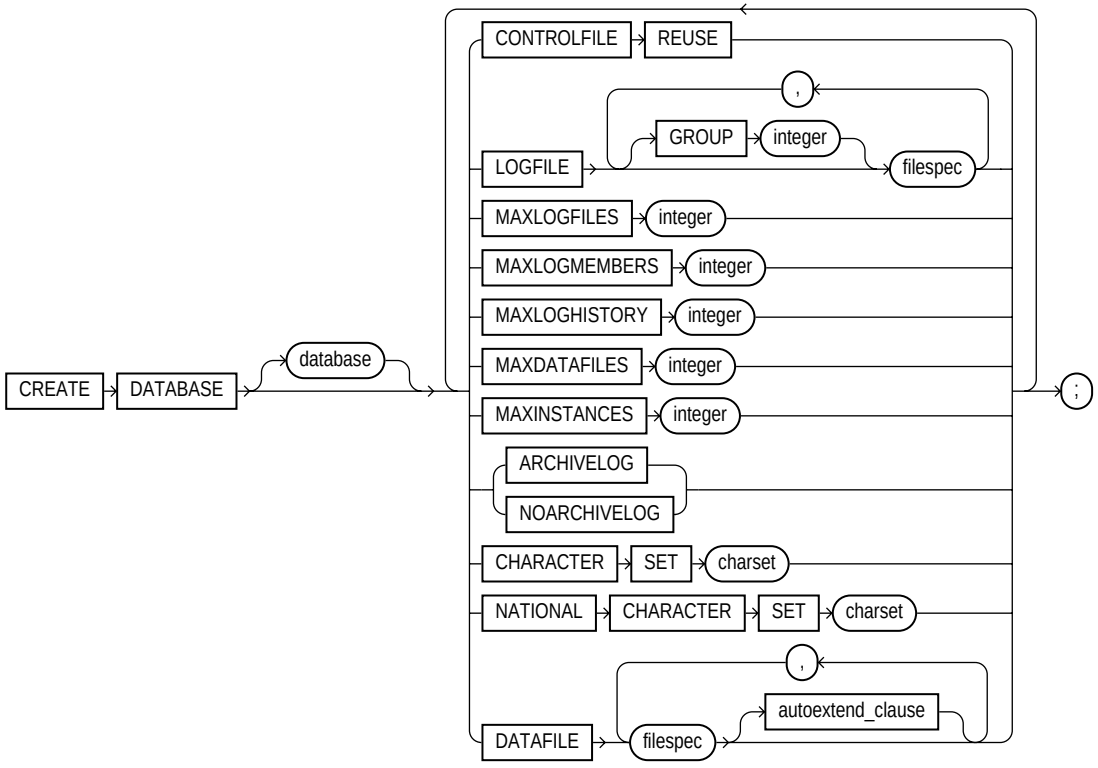
This statement re-creates a control file. In this statement, database ORDERS_2 was created with the F7DEC character set.

```
CREATE CONTROLFILE REUSE
   DATABASE orders_2
   LOGFILE GROUP 1 ('diskb:log1.log', 'diskc:log1.log') SIZE 50K,
           GROUP 2 ('diskb:log2.log', 'diskc:log2.log') SIZE 50K
   NORESETLOGS
   DATAFILE 'diska:dbone.dat' SIZE 2M
                              MAXLOGFILES 5
                              MAXLOGHISTORY 100
                              MAXDATAFILES 10
                              MAXINSTANCES 2
                              ARCHIVELOG
   CHARACTER SET F7DEC;
```

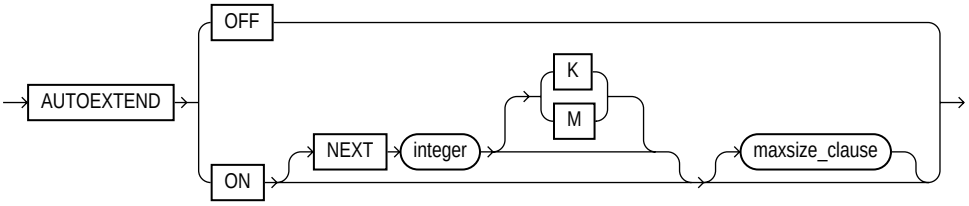
CREATE DATABASE

Syntax

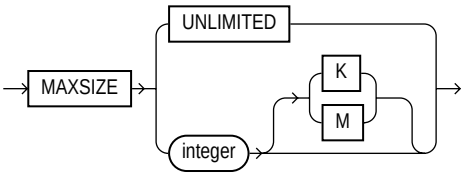
WARNING: This statement prepares a database for initial use and erases any data currently in the specified files. Use this statement only when you understand its ramifications.



autoextend_clause::=



maxsize_clause::=



filespec: See "filespec" on page 7-490.

Purpose

To create a database, making it available for general use.

This statement erases all data in any specified datafiles that already exist in order to prepare them for initial database use. If you use the statement on an existing database, all data in the datafiles is lost.

After creating the database, this statement mounts it in either exclusive or parallel mode (depending on the value of the PARALLEL_SERVER initialization parameter) and opens it, making it available for normal use. You can then create tablespaces and rollback segments for the database. For information on these tasks, see "CREATE ROLLBACK SEGMENT" on page 7-346 and "CREATE TABLESPACE" on page 7-394.

For more information on modifying a database, see "ALTER DATABASE" on page 7-6.

Prerequisites

You must have the OSDBA role enabled.

If the REMOTE_LOGIN_PASSWORDFILE initialization parameter is set to EXCLUSIVE, Oracle returns an error when you attempt to re-create the database. To avoid this message, either set the parameter to SHARED, or re-create your password file before re-creating the database. For more information about the REMOTE_LOGIN_PASSWORDFILE parameter, see *Oracle8i Reference*.

Keyword and Parameters

database is the name of the database to be created and can be up to 8 bytes long. The database name can contain only ASCII characters. Oracle writes this name into the control file. If you subsequently issue an ALTER DATABASE statement that explicitly specifies a database name, Oracle verifies that name with the name in the control file. Database names should also adhere to the rules described in "Schema Object Naming Rules" on page 2-67.

Note: You cannot use special characters from European or Asian character sets in a database name. For example, characters with umlauts are not allowed.

If you omit the database name from a CREATE DATABASE statement, Oracle uses the name specified by the initialization parameter DB_NAME. If the DB_NAME initialization parameter has been set, and you specify a different name from the value of that parameter, Oracle returns an error.

CONTROLFILE REUSE reuses existing control files identified by the initialization parameter CONTROL_FILES, thus ignoring and overwriting any information they currently contain. Normally you use this clause only when you are re-creating a database, rather than creating one for the first time. You cannot use this clause if you also specify a parameter value that requires that the control file be larger than the existing files. These parameters are MAXLOGFILES, MAXLOGMEMBERS, MAXLOGHISTORY, MAXDATAFILES, and MAXINSTANCES.

If you omit this clause and any of the files specified by CONTROL_FILES already exist, Oracle returns an error.

LOGFILE specifies one or more files to be used as redo log files. Each *filespec* specifies a redo log file group containing one or more redo log file members (copies). For the syntax of *filespec*, see "filespec" on page 7-490. All redo log files specified in a CREATE DATABASE statement are added to redo log thread number 1.

GROUP *integer* uniquely identifies a redo log file group and can range from 1 to the value of the MAXLOGFILES parameter. A database must have at least two redo log file groups. You cannot specify multiple redo log file groups having the same GROUP value. If you omit this parameter, Oracle generates its value automatically. You can examine the GROUP value for a redo log file group through the dynamic performance table V$LOG.

If you omit the LOGFILE clause, Oracle creates two redo log file groups by default. The names and sizes of the default files depend on your operating system.

MAXLOGFILES *integer*	specifies the maximum number of redo log file groups that can ever be created for the database. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The default, minimum, and maximum values depend on your operating system.
MAXLOGMEMBERS *integer*	specifies the maximum number of members, or copies, for a redo log file group. Oracle uses this value to determine how much space in the control file to allocate for the names of redo log files. The minimum value is 1. The maximum and default values depend on your operating system.
MAXLOGHISTORY *integer*	specifies the maximum number of archived redo log files for automatic media recovery with Oracle Parallel Server. Oracle uses this value to determine how much space in the control file to allocate for the names of archived redo log files. The minimum value is 0. The default value is a multiple of the MAXINSTANCES value and depends on your operating system. The maximum value is limited only by the maximum size of the control file. **Note:** This parameter is useful only if you are using Oracle with the Parallel Server option in parallel mode, and archivelog mode enabled.
MAXDATAFILES *integer*	specifies the initial sizing of the datafiles section of the control file at CREATE DATABASE or CREATE CONTROLFILE time. An attempt to add a file whose number is greater than MAXDATAFILES, but less than or equal to DB_FILES, causes the Oracle control file to expand automatically so that the datafiles section can accommodate more files. The number of datafiles accessible to your instance is also limited by the initialization parameter DB_FILES.
MAXINSTANCES *integer*	specifies the maximum number of instances that can simultaneously have this database mounted and open. This value takes precedence over the value of initialization parameter INSTANCES. The minimum value is 1. The maximum and default values depend on your operating system.
ARCHIVELOG	specifies that the contents of a redo log file group must be archived before the group can be reused. This clause prepares for the possibility of media recovery.
NOARCHIVELOG	specifies that the contents of a redo log file group need not be archived before the group can be reused. This clause does not allow for the possibility of media recovery. The default is NOARCHIVELOG mode. After creating the database, you can change between ARCHIVELOG mode and NOARCHIVELOG mode with the ALTER DATABASE statement.
CHARACTER SET	specifies the character set the database uses to store data. You cannot change the database character set after creating the database. The supported character sets and default value of this parameter depend on your operating system. **Restriction:** You cannot specify any fixed-width multibyte character sets as the database character set. For more information about character sets, see *Oracle8i National Language Support Guide*.

NATIONAL CHARACTER SET	specifies the national character set used to store data in columns specifically defined as NCHAR, NCLOB, or NVARCHAR2. If not specified, the national character set defaults to the database character set. See *Oracle8i National Language Support Guide* for valid character set names.	
DATAFILE	specifies one or more files to be used as datafiles. See the syntax description of *filespec* in "filespec" on page 7-490. All these files become part of the SYSTEM tablespace. If you omit this clause, Oracle creates one datafile by default. The name and size of this default file depend on your operating system.	
	Note: Oracle recommends that the total initial space allocated for the SYSTEM tablespace be a minimum of 5 megabytes.	
autoextend_clause	enables or disables the automatic extension of a datafile. If you do not specify this clause, datafiles are not automatically extended.	
	OFF	disables autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in ALTER DATABASE AUTOEXTEND or ALTER TABLESPACE AUTOEXTEND statements.
	ON	enables autoextend.
	NEXT	specifies the size in bytes of the next increment of disk space to be allocated to the datafile automatically when more extents are required. Use K or M to specify this size in kilobytes or megabytes. The default is the size of one data block.
	MAXSIZE	specifies the maximum disk space allowed for automatic extension of the datafile.
	UNLIMITED	sets no limit on the allocation of disk space to the datafile.

Examples

The following statement creates a small database using defaults for all arguments:

```
CREATE DATABASE;
```

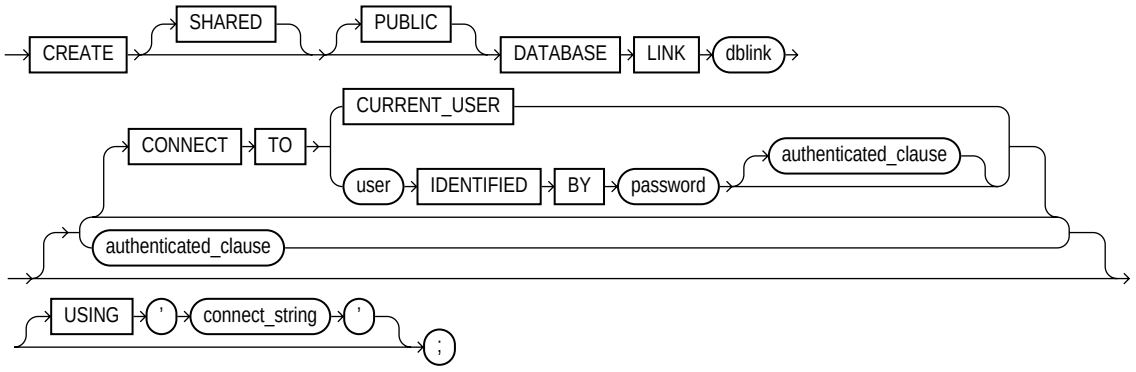
The following statement creates a database and fully specifies each argument:

```
CREATE DATABASE newtest
   CONTROLFILE REUSE
   LOGFILE
      GROUP 1 ('diskb:log1.log', 'diskc:log1.log') SIZE 50K,
      GROUP 2 ('diskb:log2.log', 'diskc:log2.log') SIZE 50K
   MAXLOGFILES 5
   MAXLOGHISTORY 100
   DATAFILE 'diska:dbone.dat' SIZE 2M
   MAXDATAFILES 10
```

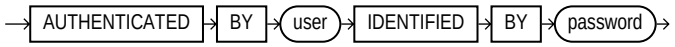
```
MAXINSTANCES 2
ARCHIVELOG
CHARACTER SET US7ASCII
NATIONAL CHARACTER SET JA16SJISFIXED
DATAFILE
'disk1:df1.dbf' AUTOEXTEND ON
'disk2:df2.dbf' AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED;
```

CREATE DATABASE LINK

Syntax



authenticated_clause::=



Purpose

To create a database link. A **database link** is a schema object in the local database that allows you to access objects on a remote database. The remote database need not be an Oracle system.

Once you have created a database link, you can use it to refer to tables and views on the remote database. You can refer to a remote table or view in a SQL statement by appending *@dblink* to the table or view name. You can query a remote table or view with the SELECT statement. If you are using Oracle with the distributed option, you can also access remote tables and views using any of the following statements:

- "DELETE" on page 7-438
- "INSERT" on page 7-512
- "LOCK TABLE" on page 7-520
- "UPDATE" on page 7-584

For information about accessing remote tables or views with PL/SQL functions, procedures, packages, and datatypes, see *Oracle8i Application Developer's Guide - Fundamentals*. For information on distributed database systems, see *Oracle8i Distributed Database Systems*.

Prerequisites

To create a private database link, you must have CREATE DATABASE LINK system privilege. To create a public database link, you must have CREATE PUBLIC DATABASE LINK system privilege.

You must have CREATE SESSION privilege on the remote Oracle database.

Net8 must be installed on both the local and remote Oracle databases.

To access non-Oracle systems you must use the Oracle Heterogeneous Services.

Keyword and Parameters

SHARED	uses a single network connection to create a public database link that can be shared between multiple users. This clause is available only with the multi-threaded server configuration. For more information about shared database links, see *Oracle8i Distributed Database Systems*.
PUBLIC	creates a public database link available to all users. If you omit this clause, the database link is private and is available only to you.
dblink	is the complete or partial name of the database link. For guidelines for naming database links, see "Referring to Objects in Remote Databases" on page 2-74. **Restrictions**: ■ You cannot create a database link in another user's schema, and you cannot qualify *dblink* with the name of a schema. (Periods are permitted in names of database links, so Oracle interprets the entire name, such as RALPH.LINKTOSALES, as the name of a database link in your schema rather than as a database link named LINKTOSALES in the schema RALPH.) ■ The number of different database links that can appear in a single statement is limited to the value of the initialization parameter OPEN_LINKS.
CONNECT TO	enables a connection to the remote database.
CURRENT_USER	creates a **current user database link**. The current user must be a global user with a valid account on the remote database for the link to succeed. If the database link is used directly, that is, not from within a stored object, then the current user is the same as the connected user.

	When executing a stored object (such as a procedure, view, or trigger) that initiates a database link, CURRENT_USER is the username that owns the stored object, and not the username that called the object. For example, if the database link appears inside procedure SCOTT.P (created by SCOTT), and user JANE calls procedure SCOTT.P, the current user is SCOTT. However, if the stored object is an invoker-rights function, procedure, or package, the invoker's authorization ID is used to connect as a remote user. For example, if the privileged database link appears inside procedure SCOTT.P (an invoker-rights procedure created by SCOTT), and user JANE calls procedure SCOTT.P, then CURRENT_USER is JANE and the procedure executes with JANE's privileges. For more information on invoker-rights functions, see "CREATE FUNCTION" on page 7-266.
user IDENTIFIED BY *password*	is the username and password used to connect to the remote database (fixed user database link). If you omit this clause, the database link uses the username and password of each user who is connected to the database (connected user database link).
authenticated_clause	specifies the username and password on the target instance. This clause authenticates the user to the remote server and is required for security. The specified username and password must be a valid username and password on the remote instance. The username and password are used only for authentication. No other operations are performed on behalf of this user. You must specify this clause when using the SHARED clause.
USING '*connect string*'	specifies the service name of a remote database. For information on specifying remote databases, see *Net8 Administrator's Guide*.

Examples

CURRENT_USER Example The following statement defines a current-user database link:

```
CREATE DATABASE LINK sales.hq.acme.com
   CONNECT TO CURRENT_USER
   USING 'sales';
```

Fixed User Example The following statement defines a fixed-user database link named SALES.HQ.ACME.COM:

```
CREATE DATABASE LINK sales.hq.acme.com
   CONNECT TO scott IDENTIFIED BY tiger
   USING 'sales';
```

Once this database link is created, you can query tables in the schema SCOTT on the remote database in this manner:

```
SELECT *
   FROM emp@sales.hq.acme.com;
```

You can also use DML statements to modify data on the remote database:

```
INSERT INTO accounts@sales.hq.acme.com(acc_no, acc_name, balance)
   VALUES (5001, 'BOWER', 2000);

UPDATE accounts@sales.hq.acme.com
   SET balance = balance + 500;

DELETE FROM accounts@sales.hq.acme.com
   WHERE acc_name = 'BOWER';
```

You can also access tables owned by other users on the same database. This statement assumes SCOTT has access to ADAM's DEPT table:

```
SELECT *
   FROM adams.dept@sales.hq.acme.com;
```

The previous statement connects to the user SCOTT on the remote database and then queries ADAM's DEPT table.

A synonym may be created to hide the fact that SCOTT's EMP table is on a remote database. The following statement causes all future references to EMP to access a remote EMP table owned by SCOTT:

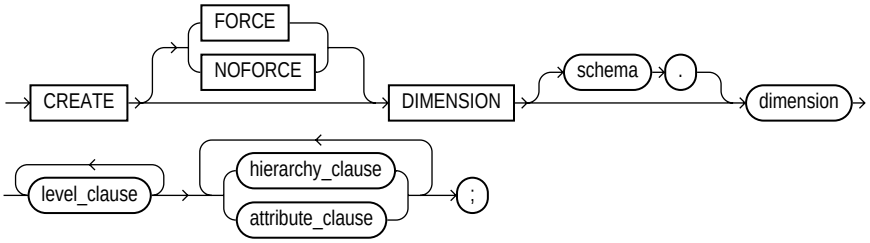
```
CREATE SYNONYM emp
   FOR scott.emp@sales.hq.acme.com;
```

PUBLIC Example The following statement defines a shared public fixed user database link named SALES.HQ.ACME.COM that refers to user SCOTT with password TIGER on the database specified by the string service name 'SALES':

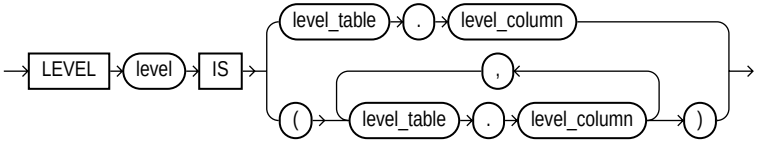
```
CREATE SHARED PUBLIC DATABASE LINK sales.hq.acme.com
   CONNECT TO scott IDENTIFIED BY tiger
   AUTHENTICATED BY anupam IDENTIFIED BY bhide
   USING 'sales';
```

CREATE DIMENSION

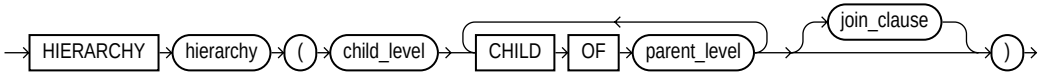
Syntax



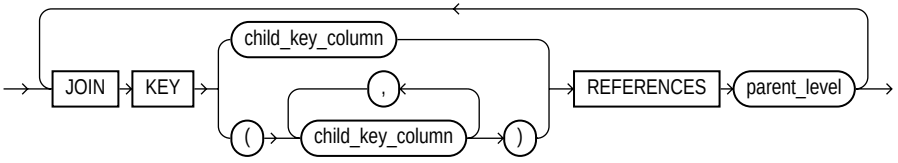
level_clause::=



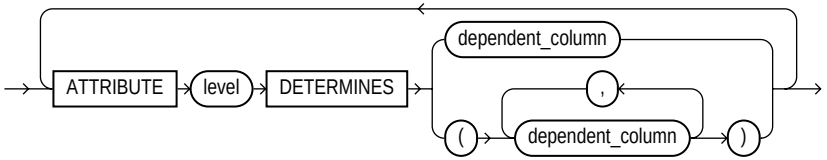
hierarchy_clause::=



join_clause::=



attribute_clause::=



Purpose

To create a **dimension**. A dimension defines a parent-child relationship between pairs of column sets, where all the columns of a column set must come from the same table. However, columns in one column set (or "level") can come from a different table than columns in another set. The optimizer uses these relationships with materialized views to perform **query rewrite**. The Summary Advisor uses these relationships to recommend creation of specific materialized views. For more information on materialized views, see "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 7-300. For more information on query rewrite, the optimizer and the Summary Advisor, see *Oracle8i Tuning*.

Prerequisites

To create a dimension in your own schema, you must have the CREATE DIMENSION system privilege. To create a dimension in another user's schema, you must have the CREATE ANY DIMENSION system privilege. In either case, you must have the SELECT object privilege on any objects referenced in the dimension.

Keywords and Parameters

FORCE	creates the dimension even if tables referenced in this statement do not yet exist, or you do not have SELECT object privilege on those tables.
	Note: Even if you specify FORCE, the optimizer cannot use this dimension for query rewrite until the tables exist and you have appropriate object privileges on them.
NOFORCE	creates the dimension only if the referenced objects exist and you have appropriate privileges on those objects. This is the default.
schema	is the schema in which the dimension will be created. If you do not specify *schema*, Oracle creates the dimension in your own schema.
dimension	is the name of the dimension. The name must be unique within its schema.

level_clause defines a level in the dimension. A level defines dimension hierarchies and attributes.

level is the name of the level

level_table . level_column specifies from the columns in the level. You can specify up to 32 columns.

Restrictions:

- All of the columns in a level must come from the same table.
- The set of columns you specify must be unique to this level.
- The columns you specify cannot be specified in any other dimension.
- Each *level_column* must be non-null. (However, these columns need not have NOT NULL constraints.)

hierarchy_clause defines a linear hierarchy of levels in the dimension. Each hierarchy forms a chain of parent-child relationships among the levels in the dimension. Hierarchies in a dimension are independent of each other. They may (but need not) have columns in common.

Each level in the dimension should be specified at most once in this clause, and each level must already have been named in the *level_clause*.

hierarchy is the name of the hierarchy. This name must be unique in the dimension.

child_level is the name of a level that has an *n*:1 relationship with a parent level: the *level_columns* of *child_level* cannot be null, and each *child_level* value uniquely determines the value of the next named *parent_level*.

If the child *level_table* is different from the parent *level_table*, you must specify a join relationship between them in the *join_clause*.

parent_level is the name of a level.

join_clause specifies an inner equijoin relationship for a dimension whose columns are contained in multiple tables. This clause is required and permitted only when the columns specified in the hierarchy are not all in the same table.

Restrictions:

- The *child_key_columns* must be non-null and the parent key must be unique and non-null. You need not define constraints to enforce these conditions, but queries may return incorrect results if these conditions are not true.
- Each child key must join with a key in the *parent_level* table.
- Self-joins are not permitted (that is, the *child_key_columns* cannot be in the same table as *parent_level*).

child_key_column specifies one or more columns that are join-compatible with columns in the parent level.

If you do not specify the schema and table of each *child_column*, the schema and table are inferred from the CHILD OF relationship in the *hierarchy_clause*. If you do specify the schema and column of a *child_key_column*, the schema and table must match the schema and table of columns that comprise the child of *parent_level* in the *hierarchy_clause*.

Restrictions:

- All of the child-key columns must come from the same table.
- The number of child-key columns must match the number of columns in *parent_level*, and the columns must be joinable.
- Do not specify multiple child key columns unless the parent level consists of multiple columns.

You can specify only one *join_clause* for a given pair of levels in the same hierarchy.

parent_level is the name of a level.

attribute_clause specifies the columns that are uniquely determined by a hierarchy level. The columns in *level* must all come from the same table as the *dependent_columns*. The *dependent_columns* need not have been specified in the *level_clause*.

For example, if the hierarchy levels are *city*, *state*, and *country*, then *city* might determine *mayor*, *state* might determine *governor*, and *country* might determine *president*.

Examples

This statement creates a TIME dimension on table TIME_TAB, and creates a GEOG dimension on tables CITY, STATE, and COUNTRY.

```
CREATE DIMENSION time
   LEVEL curDate         IS time_tab.curDate
   LEVEL month           IS time_tab.month
   LEVEL qtr             IS time_tab.qtr
   LEVEL year            IS time_tab.year
   LEVEL fiscal_week     IS time_tab.fiscal_week
   LEVEL fiscal_qtr      IS time_tab.fiscal_qtr
   LEVEL fiscal_year     IS time_tab.fiscal_year
   HIERARCHY month_rollup (
      curDate            CHILD OF
      month              CHILD OF
      qtr                CHILD OF
      year)
   HIERARCHY fiscal_year_rollup (
      curDate            CHILD OF
```

```
      fiscal_week        CHILD OF
      fiscal_qtr         CHILD OF
      fiscal_year )
   ATTRIBUTE curDate     DETERMINES (holiday, dayOfWeek)
   ATTRIBUTE month       DETERMINES (yr_ago_month, qtr_ago_month)
   ATTRIBUTE fiscal_qtr  DETERMINES yr_ago_qtr
   ATTRIBUTE year        DETERMINES yr_ago ;

CREATE DIMENSION geog
   LEVEL cityID         IS (city.city, city.state)
   LEVEL stateID        IS state.state
   LEVEL countryID      IS country.country
   HIERARCHY political_rollup (
      cityID            CHILD OF
      stateID           CHILD OF
      countryID
         JOIN KEY city.state REFERENCES stateID
         JOIN KEY state.country REFERENCES countryID);
```

CREATE DIRECTORY

Syntax

Purpose

To create a directory object. A directory object specifies an alias for a directory on the server's file system where external binary file LOBs (BFILEs) are located. You can use directory names when referring to BFILEs in your PL/SQL code and OCI calls, rather than hard-coding the operating system pathname, thereby allowing greater file management flexibility. For more information on BFILE objects, see "Large Object (LOB) Datatypes" on page 2-19.

All directories are created in a single namespace and are not owned by an individual's schema. You can secure access to the BFILEs stored within the directory structure by granting object privileges on the directories to specific users. for more information on granting object privileges, see "Large Object (LOB) Datatypes" on page 2-19.

When you create a directory, you are automatically granted the READ object privilege and can grant READ privileges to other users and roles. The DBA can also grant this privilege to other users and roles.

Prerequisites

You must have CREATE ANY DIRECTORY system privileges to create directories.

You must also create a corresponding operating system directory for file storage. Your system or database administrator must ensure that the operating system directory has the correct read permissions for Oracle processes.

Privileges granted for the directory are created independently of the permissions defined for the operating system directory. Therefore, the two may or may not correspond exactly. For example, an error occurs if user SCOTT is granted READ privilege on the directory schema object, but the corresponding operating system directory does not have READ permission defined for Oracle processes.

Keywords and Parameters

OR REPLACE	re-creates the directory database object if it already exists. You can use this clause to change the definition of an existing directory without dropping, re-creating, and regranting database object privileges previously granted on the directory. Users who had previously been granted privileges on a redefined directory can still access the directory without being regranted the privileges For information on removing a directory from the database, see "DROP DIRECTORY" on page 7-451.
directory	is the name of the directory object to be created. The maximum length of *directory* is 30 bytes. You cannot qualify a directory object with a schema name. **Note:** Oracle does not verify that the directory you specify actually exists. Therefore, take care that you specify a valid directory in your operating system. In addition, if your operating system uses case-sensitive pathnames, be sure you specify the directory in the correct format. (However, you need not include a trailing slash at the end of the pathname.)
'path_name'	is the full pathname of the operating system directory on the server where the files are located. The single quotes are required, with the result that the path name is case sensitive.

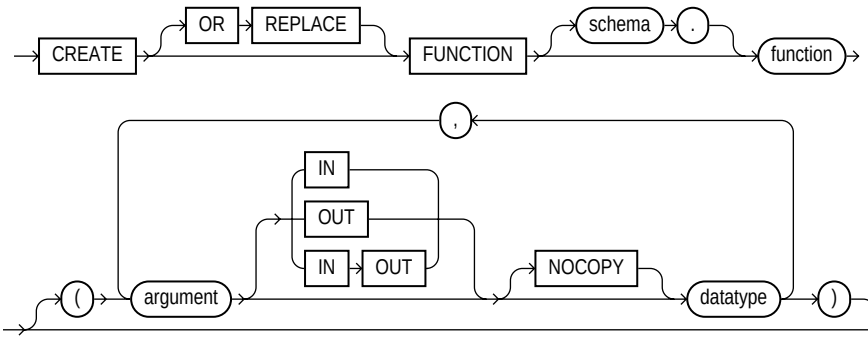
Example

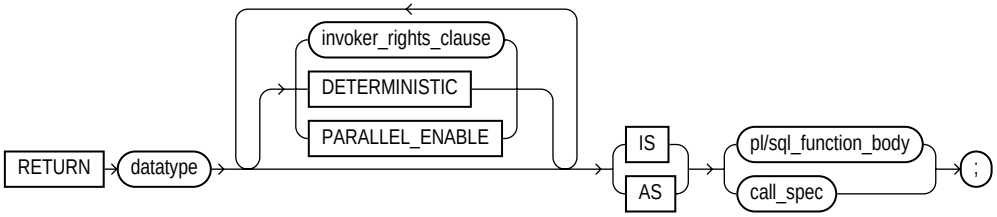
The following statement redefines directory database object BFILE_DIR to enable access to BFILEs stored in the operating system directory /PRIVATE1/LOB/FILES:

```
CREATE OR REPLACE DIRECTORY bfile_dir AS '/private1/LOB/files';
```

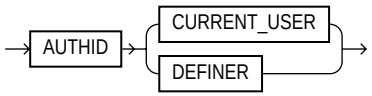
CREATE FUNCTION

Syntax

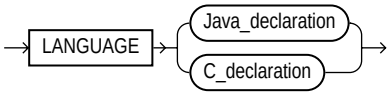




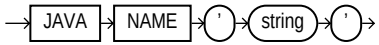
invoker_rights_clause::=



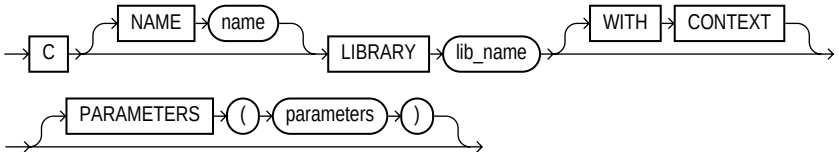
call_spec::=



Java_declaration::=



C_declaration::=



Purpose

To create a stored function or a call specification.

A **stored function** (also called a **user function**) is a set of PL/SQL statements you can call by name. Stored functions are very similar to procedures, except that a function returns a value to the environment in which it is called. User functions can be used as part of a SQL expression. For a general discussion of procedures and functions, see "CREATE PROCEDURE" on page 7-333. For examples of creating functions, see "Examples" on page 7-271.

A **call specification** declares a Java method or a third-generation language (3GL) routine so that it can be called from SQL and PL/SQL. The call specification tells Oracle which Java method, or which named function in which shared library, to invoke when a call is made. It also tells Oracle what type conversions to make for the arguments and return value.

The CREATE FUNCTION statement creates a function as a standalone schema object. You can also create a function as part of a package. For information on creating packages, see "CREATE PACKAGE" on page 7-325.

For information on modifying a function, see "ALTER FUNCTION" on page 7-27. For information on shared libraries, see "CREATE LIBRARY" on page 7-298. For information on dropping a standalone function, see "DROP FUNCTION" on page 7-452. For more information about registering external functions, see *Oracle8i Application Developer's Guide - Fundamentals*.

Prerequisites

Before a stored function can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

To create a function in your own schema, you must have the CREATE PROCEDURE system privilege. To create a function in another user's schema, you must have the CREATE ANY PROCEDURE system privilege. To replace a function

in another user's schema, you must have the ALTER ANY PROCEDURE system privilege.

To invoke a call specification, you may need additional privileges (for example, EXECUTE privileges on C library for a C call specification). For more information on such prerequisites, refer to *PL/SQL User's Guide and Reference* or *Oracle8i Java Stored Procedures Developer's Guide.*

To embed a CREATE FUNCTION statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

Keywords and Parameters

OR REPLACE — re-creates the function if it already exists. Use this clause to change the definition of an existing function without dropping, re-creating, and regranting object privileges previously granted on the function. If you redefine a function, Oracle recompiles it. For information on recompiling functions, see "ALTER FUNCTION" on page 7-27.

Users who had previously been granted privileges on a redefined function can still access the function without being regranted the privileges.

If any function-based indexes depend on the function, Oracle marks the indexes DISABLED.

schema — is the schema to contain the function. If you omit *schema*, Oracle creates the function in your current schema.

function — is the name of the function to be created. If creating the function results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SHOW ERRORS command.

Restrictions on User-Defined Functions

User-defined functions cannot be used in situations that require an unchanging definition. Thus, you cannot use user-defined functions:

- In a CHECK constraint clause of a CREATE TABLE or ALTER TABLE statement
- In a DEFAULT clause of a CREATE TABLE or ALTER TABLE statement

In addition, when a function is called from within a query or DML statement, the function cannot:

- Have OUT or IN OUT parameters
- Commit or roll back the current transaction, create or roll back to a savepoint, or alter the session or the system. DDL statements implicitly commit the current transaction, so a user-defined function cannot execute any DDL statements.
- Write to the database, if the function is being called from a SELECT statement. However, a function called from a subquery in a DML statement can write to the database.
- Write to the same table that is being modified by the statement from which the function is called, if the function is called from a DML statement.

Except for the restriction on OUT and IN OUT parameters, Oracle enforces these restrictions not only for the function called directly from the SQL statement, but also for any functions that function calls, and on any functions called from the SQL statements executed by that function or any function it calls.

argument is the name of an argument to the function. If the function does not accept arguments, you can omit the parentheses following the function name.

IN specifies that you must supply a value for the argument when calling the function. This is the default.

OUT specifies the function will set the value of the argument.

IN OUT specifies that a value for the argument can be supplied by you and may be set by the function.

NOCOPY instructs Oracle to pass this argument as fast as possible. This clause can significantly enhance performance when passing a large value like a record, a PL/SQL table, or a varray to an OUT or IN OUT parameter. (IN parameter values are always passed NOCOPY.)

- When you specify NOCOPY, assignments made to a package variable may show immediately in this parameter (or assignments made to this parameter may show immediately in a package variable) if the package variable is passed as the actual assignment corresponding to this parameter.
- Similarly, changes made either to this parameter or to another parameter may be visible immediately through both names if the same variable is passed to both.
- If the function is exited with an unhandled exception, any assignment made to this parameter may be visible in the caller's variable.

These effects may or may not occur on any particular call. You should use NOCOPY only when these effects would not matter.

datatype is the datatype of an argument. An argument can have any datatype supported by PL/SQL.

The datatype cannot specify a length, precision, or scale. Oracle derives the length, precision, or scale of an argument from the environment from which the function is called.

RETURN *datatype* specifies the datatype of the function's return value. Because every function must return a value, this clause is required. The return value can have any datatype supported by PL/SQL.

The datatype cannot specify a length, precision, or scale. Oracle derives the length, precision, or scale of the return value from the environment from which the function is called. For information on PL/SQL datatypes, see *PL/SQL User's Guide and Reference.*

invoker_rights_clause lets you specify whether the function executes with the privileges and in the schema of the user who owns it or with the privileges and in the schema of CURRENT_USER. For information on how CURRENT_USER is determined, see *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals.*

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the function. For more information refer to *PL/SQL User's Guide and Reference*.

AUTHID CURRENT_USER specifies that the function executes with the privileges of CURRENT_USER. This clause creates an "invoker-rights function."

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the function resides.

AUTHID DEFINER specifies that the function executes with the privileges of the owner of the schema in which the function resides, and that external names resolve in the schema where the function resides. This is the default.

DETERMINISTIC is an optimization hint that allows the system to use a saved copy of the function's return result (if such a copy is available). The saved copy could come from a materialized view, a function-based index, or a redundant call to the same function in the same SQL statement. The query optimizer can choose whether to use the saved copy or re-call the function.

The function should reliably return the same result value whenever it is called with the same values for its arguments. Therefore, do not define the function to use package variables or to access the database in any way that might affect the function's return result, because the results of doing so will not be captured if the system chooses not to call the function.

A function must be declared DETERMINISTIC in order to be called in the expression of a function-based index, or from the query of a materialized view if that view is marked REFRESH FAST or ENABLE QUERY REWRITE.

For information on materialized views, see *Oracle8i Tuning.* For information on function-based indexes, see "CREATE INDEX" on page 7-273.

PARALLEL_ENABLE is an optimization hint indicating that the function can be executed from a parallel execution server of a parallel query operation. The function should not use session state, such as package variables, as those variables may not be shared among the parallel execution servers. For more information on these concepts, see *Oracle8i Application Developer's Guide - Fundamentals.*

pl/sql_ subprogram_body	declares the function in a PL/SQL subprogram body. For more information on PL/SQL subprograms, see *Oracle8i Application Developer's Guide - Fundamentals*.
call_spec	maps a Java or C method name, parameter types, and return type to their SQL counterparts. ■ In *Java_declaration*, '*string*' identifies the Java implementation of the method. For more information, see *Oracle8i Java Stored Procedures Developer's Guide*. ■ For an explanation of the parameters and semantics of *the C_declaration*, see *Oracle8i Application Developer's Guide - Fundamentals*. AS EXTERNAL is an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle Corporation recommends that you use the AS LANGUAGE C syntax.

Examples

The following statement creates the function GET_BAL.

```
CREATE FUNCTION get_bal(acc_no IN NUMBER)
   RETURN NUMBER
   IS acc_bal NUMBER(11,2);
   BEGIN
      SELECT balance
      INTO acc_bal
      FROM accounts
      WHERE account_id = acc_no;
      RETURN(acc_bal);
    END;
```

The GET_BAL function returns the balance of a specified account.

When you call the function, you must specify the argument ACC_NO, the number of the account whose balance is sought. The datatype of ACC_NO is NUMBER.

The function returns the account balance. The RETURN clause of the CREATE FUNCTION statement specifies the datatype of the return value to be NUMBER.

The function uses a SELECT statement to select the BALANCE column from the row identified by the argument ACC_NO in the ACCOUNTS table. The function uses a RETURN statement to return this value to the environment in which the function is called.

The function created above can be used in a SQL statement. For example:

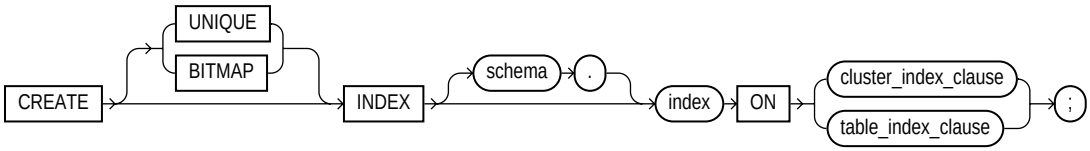
```
SELECT get_bal(100) FROM DUAL;
```

The following statement creates PL/SQL standalone function GET_VAL that registers the C routine C_GET_VAL as an external function. (The parameters have been omitted from this example.)

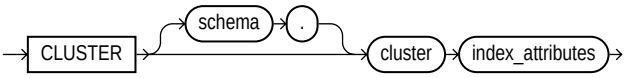
```
CREATE FUNCTION get_val
   ( x_val IN NUMBER,
   y_val IN NUMBER,
   image IN LONG RAW )
   RETURN BINARY_INTEGER AS LANGUAGE C
      NAME "c_get_val"
      LIBRARY c_utils
      PARAMETERS (...);
```

CREATE INDEX

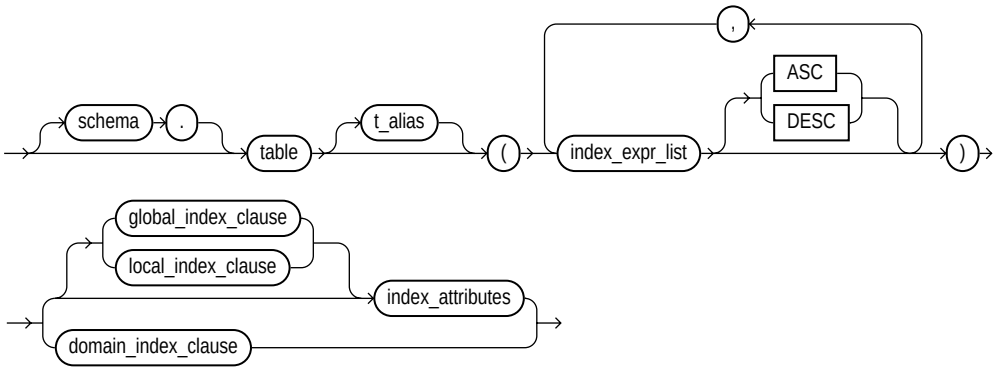
Syntax



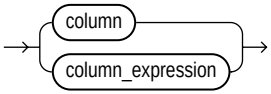
cluster_index_clause::=



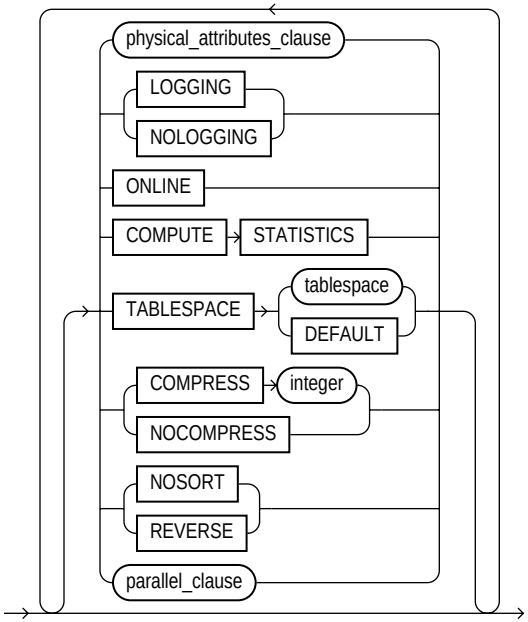
table_index_clause::=



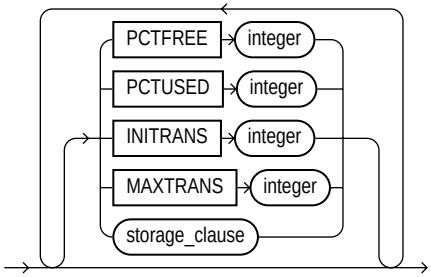
index_expr_list::=



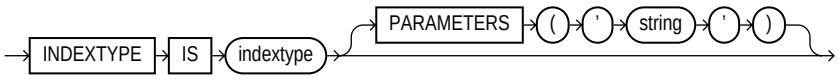
index_attributes::=



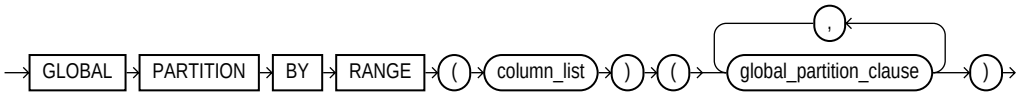
physical_attributes_clause::=



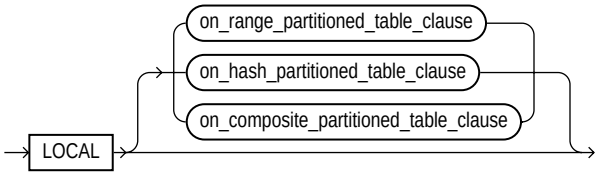
domain_index_clause::=



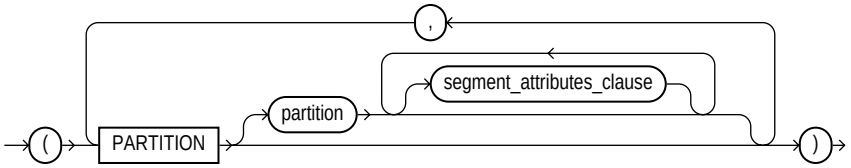
global_index_clause::=



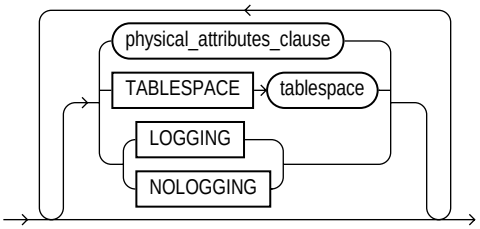
local_index_clauses::=



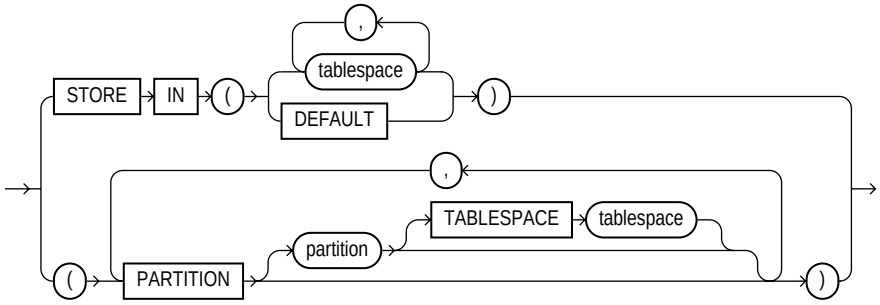
on_range_partitioned_table_clause::=



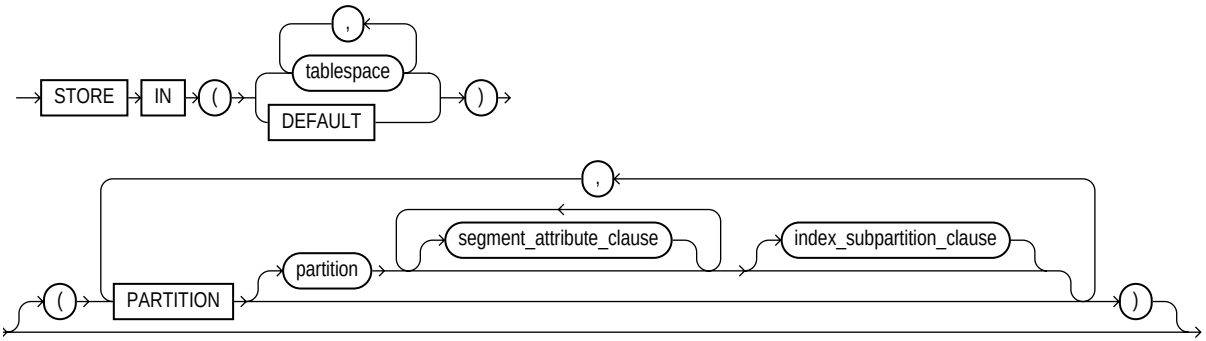
segment_attributes_clause::=



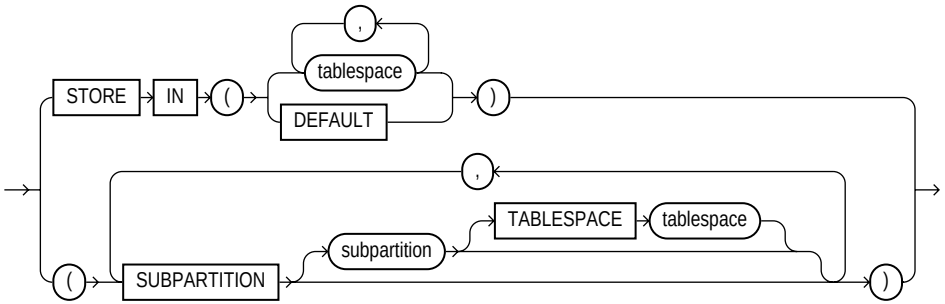
on_hash_partitioned_table_clause::=



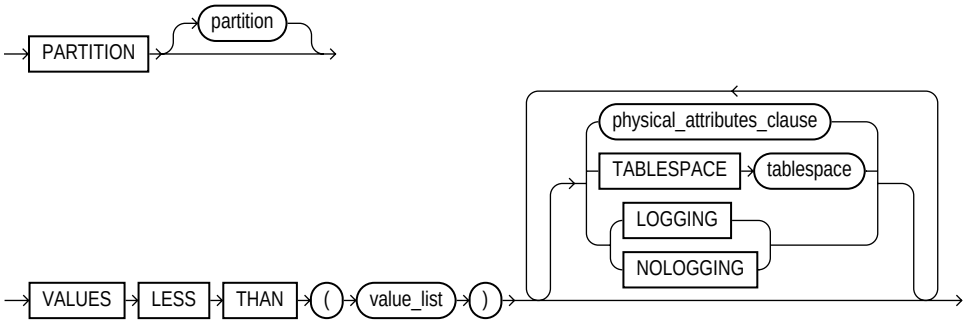
on_composite_partitioned_table_clause::=



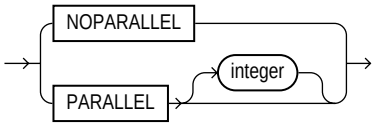
index_subpartition_clause::=



global_partition_clause::=



parallel_clause::=



storage_clause: See "storage_clause" on page 7-575.

Purpose

To create an index on

- One or more columns of a table, a partitioned table, an index-organized table, or a cluster
- One or more scalar typed object attributes of a table or a cluster
- A nested table storage table for indexing a nested table column

To create a **domain index**, which is an instance of an application-specific index of type *indextype*.

An **index** is a schema object that contains an entry for each value that appears in the indexed column(s) of the table or cluster and provides direct, fast access to rows. A **partitioned index** consists of partitions containing an entry for each value that appears in the indexed column(s) of the table. A **function-based index** is an index on expressions. It enables you to construct queries that evaluate the value returned by an expression, which in turn may include functions (built-in or user-defined).

For a discussion of indexes, see *Oracle8i Concepts*. For information on modifying an index, see "ALTER INDEX" on page 7-29.

Prerequisites

To create an index in your own schema, one of the following conditions must be true:

- The table or cluster to be indexed must be in your own schema.
- You must have INDEX privilege on the table to be indexed.
- You must have CREATE ANY INDEX system privilege.

To create an index in another schema, you must have CREATE ANY INDEX system privilege. Also, the owner of the schema to contain the index must have either space quota on the tablespaces to contain the index or index partitions, or UNLIMITED TABLESPACE system privilege.

To create a domain index in your own schema, you must also have EXECUTE privilege on the indextype. If you are creating a domain index in another user's schema, the index owner also must have EXECUTE privilege on the indextype and its underlying implementation type. Before creating a domain index, you should first define the indextype. See "CREATE INDEXTYPE" on page 7-291.

To create a function-based index in your own schema on your own table, you must have the QUERY REWRITE system privilege. To create the index in another schema or on another schema's table, you must have the GLOBAL QUERY REWRITE privilege. The table owner must also have the EXECUTE object privilege on the function(s) used in the function-based index.

Keywords and Parameters

UNIQUE

specifies that the value of the column (or columns) upon which the index is based must be unique. If the index is local nonprefixed (see *local_index_clause* below), then the index key must contain the partitioning key.

Oracle recommends that you do not explicitly define UNIQUE indexes on tables. Uniqueness is strictly a logical concept and should be associated with the *definition* of a table. Therefore, define UNIQUE integrity constraints on the desired columns. For more information on constraints, see "constraint_clause" on page 7-217.

Restrictions:

- You cannot specify both UNIQUE and BITMAP.
- You cannot specify UNIQUE for a domain index.

BITMAP — specifies that *index* is to be created as a bitmap, rather than as a B-tree. Bitmap indexes store the rowids associated with a key value as a bitmap. Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value. The internal representation of bitmaps is best suited for applications with low levels of concurrent transactions, such as data warehousing. See *Oracle8i Concepts* and *Oracle8i Tuning* for more information about using bitmap indexes.

Restrictions:

- You cannot specify BITMAP when creating a global partitioned index or an index-organized table.
- You cannot specify both UNIQUE and BITMAP.
- You cannot specify BITMAP for a domain index.

schema — is the schema to contain the index. If you omit *schema*, Oracle creates the index in your own schema.

index — is the name of the index to be created. An *index* can contain several partitions.

cluster_index_clause — specifies the cluster for which a cluster index is to be created. If you do not qualify cluster with *schema*, Oracle assumes the cluster is in your current schema. You cannot create a cluster index for a hash cluster. For more information on clusters, see "CREATE CLUSTER" on page 7-236.

table_index_clause — specifies *table* (and its attributes) on which you are defining the index. If you do not qualify *table* with *schema*, Oracle assumes the table is contained in your own schema.

You create an index on a nested table column by creating the index on the nested table storage table. Include the NESTED_TABLE_ID pseudocolumn of the storage table to create a UNIQUE index, which effectively ensures that the rows of a nested table value are distinct.

Restrictions:

- If the index is local, then *table* must be partitioned.
- If the table is index-organized, this statement creates a secondary index. You cannot specify BITMAP or REVERSE for this secondary index, and the combined size of the index key and the logical rowid should be less than half the block size.
- If *table* is a temporary table, the index will also be temporary with the same scope (session or transaction) as *table*. The following restrictions apply to indexes on temporary table:

  - The index cannot be a partitioned index or a domain index.

  - You cannot specify the *physical_attributes_clause* or the *parallel_clause*.

  - You cannot specify LOGGING, NOLOGGING, or TABLESPACE.

  For more information on temporary tables, see "CREATE TABLE" on page 7-359 and *Oracle8i Concepts*.

t_alias — specifies a correlation name (alias) for the table upon which you are building the index.

Note: This alias is required if the *index_expression_list* references any object type attributes or object type methods. See "Function-based Index on Type Method Example" on page 7-287.

index_expr_list lets you specify the column or column expression upon which the index is based.

column is the name of a column in the table. A bitmap index can have a maximum of 30 columns. Other indexes can have as many as 32 columns.

Restriction: You cannot create an index on columns or attributes whose type is user-defined, LONG, LONG RAW, LOB, or REF, except that Oracle supports an index on REF type columns or attributes that have been defined with a SCOPE clause.

You can create an index on a scalar object attribute column or on the system-defined NESTED_TABLE_ID column of the nested table storage table. If you specify an object attribute column, the column name must be qualified with the table name. If you specify a nested table column attribute, it must be qualified with the outermost table name, the containing column name, and all intermediate attribute names leading to the nested table column attribute.

column_expression is an expression built from columns of *table*, constants, SQL functions, and user-defined functions. When you specify *column_expression*, you create a **function-based index**.

Name resolution of the function is based on the schema of the index creator. User-defined functions used in *column_expression* are fully name resolved during the CREATE INDEX operation.

After creating a function-based index, collect statistics on both the index and its base table using the ANALYZE statement (see "ANALYZE" on page 7-185). Oracle cannot use the function-based index until these statistics have been generated.

When you subsequently query a table that uses a function-based index, you must ensure in the query that *column_expression* is not null. See the Function-Based Index Example on page 7-287.

If the function on which the index is based becomes invalid or is dropped, Oracle marks the index DISABLED. Queries on a DISABLED index fail if the optimizer chooses to use the index. DML operations on a DISABLED index fail unless

- The index is also marked UNUSABLE **and**
- The parameter `SKIP_UNUSABLE` is set to true (see "ALTER SESSION" on page 7-78 and "ALTER SYSTEM" on page 7-95 for more information on this parameter).

Oracle's use of function-based indexes is also affected by the setting of the `QUERY_REWRITE_ENABLED` session parameter. For more information, see "ALTER SESSION" on page 7-78.

Restrictions on function-based indexes:

- Any user-defined function referenced in *column_expression* must be DETERMINISTIC. For more information, see "CREATE FUNCTION" on page 7-266 and *PL/SQL User's Guide and Reference*.
- For a function-based globally partitioned index, the *column_expression* cannot be the partitioning key.
- All functions must be specified with parentheses, even if they have no parameters. Otherwise Oracle interprets them as column names.
- Any function you specify in *column_expression* must return a repeatable value. For example, you cannot specify the SYSDATE or USER function or the ROWNUM pseudocolumn.
- You cannot build a function-based index on LOB, REF, nested table, or varray columns. In addition, the function in *column_expression* cannot take as arguments any objects with attributes of type LOB, REF, nested table, or varray.
- The *column_expression* cannot contain any aggregate functions.

Note: If a public synonym for a function, package, or type is used in *column_expression*, and later an actual object with the same name is created in the table owner's schema, then Oracle will disable the function-based index. When you subsequently enable the function-based index using ALTER INDEX ... ENABLE or ALTER INDEX ... REBUILD, the function, package, or type used in the *column_expression* will continue to resolve to the function, package, or type to which the public synonym originally pointed. It will not resolve to the new function, package, or type.

ASC | DESC — specifies whether the index should be created in ascending or descending order. Indexes on character data are created in ascending or descending order of the character values in the database character set.

Oracle treats descending indexes as if they were function-based indexes. You do not need the QUERY REWRITE or GLOBAL QUERY REWRITE privileges to create them, as you do with other function-based indexes. However, as with other function-based indexes, Oracle does not use descending indexes until you first analyze the index and the table on which the index is defined. See the *column_expression* clause of this statement.

Restriction: You cannot specify either of these clauses for a domain index. You cannot specify DESC for a bitmapped index or a reverse index.

index_attributes

physical_attributes_clause — establishes values for physical and storage characteristics for the index. See "CREATE TABLE" on page 7-359.

Restriction: You cannot specify the PCTUSED parameter for an index.

PCTFREE — is the percentage of space to leave free for updates and insertions within each of the index's data blocks.

storage_clause — establishes the storage characteristics for the index. See the "storage_clause" on page 7-575.

TABLESPACE — is the name of the tablespace to hold the index, index partition, or index subpartition. If you omit this clause, Oracle creates the index in the default tablespace of the owner of the schema containing the index.

For a local index, you can specify the keyword DEFAULT in place of *tablespace*. New partitions or subpartitions added to the local index will be created in the same tablespace(s) as the corresponding partitions or subpartitions of the underlying table.

COMPRESS — enables key compression, which eliminates repeated occurrence of key column values and may substantially reduce storage. Use *integer* to specify the prefix length (number of prefix columns to compress).

- For unique indexes, the valid range of prefix length values is from 1 to the number of key columns minus 1. The default prefix length is the number of key columns minus 1.
- For nonunique indexes, the valid range of prefix length values is from 1 to the number of key columns. The default prefix length is number of key columns.

Oracle compresses only nonpartitioned indexes that are nonunique or unique indexes of at least two columns.

Restriction: You cannot specify COMPRESS for a bitmapped index.

NOCOMPRESS — disables key compression. This is the default.

NOSORT — indicates to Oracle that the rows are stored in the database in ascending order, so that Oracle does not have to sort the rows when creating the index. If the rows of the indexed column or columns are not stored in ascending order, Oracle returns an error. For greatest savings of sort time and space, use this clause immediately after the initial load of rows into a table.

Restrictions:

- You cannot specify REVERSE with this clause.
- You cannot use this clause to create a cluster, partitioned, or bitmap index.
- You cannot specify this clause for a secondary index on an index-organized table.

REVERSE — stores the bytes of the index block in reverse order, excluding the rowid. You cannot specify NOSORT with this clause.

You cannot reverse a bitmap index or an index-organized table.

LOGGING | NOLOGGING — specifies that the creation of the index will be logged (LOGGING) or not logged (NOLOGGING) in the redo log file. It also specifies that subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against the index are logged or not logged. LOGGING is the default.

If *index* is nonpartitioned, this is the logging attribute of the index.

If *index* is partitioned, the logging attribute specified is

- The default value of all partitions specified in the CREATE statement (unless you specify LOGGING | NOLOGGING in the PARTITION description clause)
- The default value for the segments associated with the index partitions
- The default value for local index partitions or subpartitions added implicitly during subsequent ALTER TABLE ... ADD PARTITION operations

In NOLOGGING mode, data is modified with minimal logging (to mark new extents INVALID and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, since the redo data is not logged. Thus if you cannot afford to lose this index, it is important to take a backup after the NOLOGGING operation.

If the database is run in ARCHIVELOG mode, media recovery from a backup taken before the LOGGING operation will re-create the index. However, media recovery from a backup taken before the NOLOGGING operation will not re-create the index.

The logging attribute of the index is independent of that of its base table.

If you omit this clause, the logging attribute is that of the tablespace in which it resides.

For more information about logging and parallel DML, see *Oracle8i Concepts* and *Oracle8i Parallel Server Concepts and Administration*.

ONLINE — specifies that DML operations on the table will be allowed during creation of the index. For a description of online index building and rebuilding, see *Oracle8i Concepts*.

Restriction: Parallel DML is not supported during online index building. If you specify ONLINE and then issue parallel DML statements, Oracle returns an error.

COMPUTE STATISTICS — enables you to collect statistics at relatively little cost during the creation of an index. These statistics are stored in the data dictionary for ongoing use by the optimizer in choosing a plan of execution for SQL statements.

The types of statistics collected depend on the type of index you are creating.

Note: If you create an index using another index (instead of a table), the original index might not provide adequate statistical information. Therefore, Oracle generally uses the base table to compute the statistics, which will improve the statistics but may negatively affect performance.

Additional methods of collecting statistics are available in PL/SQL packages and procedures. For more information, refer to *Oracle8i Supplied Packages Reference*.

global_index_clause — specifies that the partitioning of the index is user defined and is not equipartitioned with the underlying table. By default, nonpartitioned indexes are global indexes.

PARTITION BY RANGE — specifies that the global index is partitioned on the ranges of values from the columns specified in *column_list*. You cannot specify this clause for a local index.

(column_list) — is the name of the column(s) of a table on which the index is partitioned. The *column_list* must specify a left prefix of the index column list.

You cannot specify more than 32 columns in *column_list*, and the columns cannot contain the ROWID pseudocolumn or a column of type ROWID.

PARTITION *partition* — describes the individual partitions. The number of clauses determines the number of partitions. If you omit *partition*, Oracle generates a name with the form SYS_P*n*.

VALUES LESS THAN (*value_list*) — specifies the (noninclusive) upper bound for the current partition in a global index. The *value_list* is a comma-separated, ordered list of literal values corresponding to *column_list* in the *partition_by_range_clause*. Always specify MAXVALUE as the *value_list* of the last partition.

Note: If *index* is partitioned on a DATE column, and if the NLS date format does not specify the century with the year, you must use the TO_DATE function with a 4-character format mask for the year. The NLS date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT. For more information on these initialization parameters, see *Oracle8i National Language Support Guide*. See also the "Partitioned Table Example" on page 7-389.

Restriction: You cannot specify this clause for a local index.

local_index_clauses — specify that the index is partitioned on the same columns, with the same number of partitions and the same partition bounds as *table*. Oracle automatically maintains LOCAL index partitioning as the underlying table is repartitioned.

on_range_partitioned_table_clause — describes an index on a range-partitioned table.

PARTITION *partition* — describes the individual partitions. The number of clauses determines the number of partitions. For a local index, the number of index partitions must be equal to the number of the table partitions, and in the same order.

If you omit *partition*, Oracle generates a name that is consistent with the corresponding table partition. If the name conflicts with an existing index partition name, the form SYS_P*n* is used.

on_hash_partitioned_table_clause — describes an index on a hash-partitioned table. If you do not specify *partition*, Oracle uses the name of the corresponding base table partition, unless it conflicts with an explicitly specified name of another index partition. In this case, Oracle generates a name of the form SYS_P*nnn*.

You can optionally specify TABLESPACE for all index partitions or for one or more individual partitions. If you do not specify TABLESPACE at the index or partition level, Oracle stores each index partition in the same tablespace as the corresponding table partition.

on_composite_partitioned_table_clause describes an index on a composite-partitioned table. The first STORE IN clause specifies the default tablespace for the index subpartitions. You can override this storage by specifying a different tablespace in the *index_subpartitioning_clause*.

If you do not specify TABLESPACE for subpartitions either in this clause or in the *index_subpartitioning_clause*, Oracle uses the tablespace specified for *index*. If you also do not specify TABLESPACE for *index*, Oracle stores the subpartition in the same tablespace as the corresponding table subpartition.

STORE IN lets you specify how index hash partitions (for a hash-partitioned index) or index subpartitions (for a composite-partitioned index) are to be distributed across various tablespaces. The number of tablespaces does not have to equal the number of index partitions. If the number of index partitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.

DEFAULT is valid only for a local index on a hash or composite-partitioned table. This clause overrides any tablespace specified at the index level for a partition or subpartition, and stores the index partition or subpartition in the same partition as the corresponding table partition or subpartition.

index_subpartition_clause specifies one or more tablespaces in which to store all subpartitions in *partition* or one or more individual subpartitions in *partition*. The subpartition inherits all other attributes from *partition*. Attributes not specified for *partition* are inherited from *index*.

domain_index_clause specifies that index is a domain index.

Restrictions:

- The *index_expr_list* can specify only a single column.
- You can define only one domain index on a column.
- You cannot specify a BITMAP, UNIQUE, or function-based domain index.
- You cannot create a local domain index on a partitioned table.
- You cannot create a domain index on a partitioned table with row movement enabled.

column specifies the table columns or object attributes on which the index is defined. Each *column* can have only one domain index defined on it.

Restrictions:

- You cannot create a domain index on a column of datatype REF, varray, nested table, LONG, or LONG RAW.
- You can create a domain index on a column of user-defined type, but not on an attribute of a column of user-defined type if that attribute itself is a user-defined type.

	indextype	specifies the name of the indextype. This name should be a valid schema object that you have already defined. See "CREATE INDEXTYPE" on page 7-291.
	PARAMETERS '*string*'	specifies the parameter string that is passed uninterrupted to the appropriate indextype routine. The maximum length of the parameter string is 1000 characters. Once the domain index is created, Oracle invokes this routine (see *Oracle8i Data Cartridge Developer's Guide* for information on these routines.) If the routine does not return successfully, the domain index is marked FAILED. The only operation supported on an failed domain index is DROP INDEX.
parallel_clause	causes creation of the index to be parallelized. For additional information, see the Notes to the *parallel_clause* of "CREATE TABLE" on page 7-359.	
	NOPARALLEL	specifies serial execution. This is the default.
	PARALLEL	causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.
	PARALLEL *integer*	specifies the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Examples

PARALLEL Example The following statement creates an index using 10 parallel execution servers, 5 to scan SCOTT.EMP and another 5 to populate the EMP_IDX index:

```
CREATE INDEX emp_idx
   ON scott.emp (ename)
   PARALLEL 5;
```

COMPRESS Example To create an index with the COMPRESS clause, you might issue the following statement:

```
CREATE INDEX emp_idx2 ON emp(job, ename) COMPRESS 1;
```

The index will compress repeated occurrences of JOB column values.

NOLOGGING Example To quickly create an index in parallel on a table that was created using a fast parallel load (so all rows are already sorted), you might issue the following statement. (Oracle will choose the appropriate degree of parallelism.)

```
CREATE INDEX i_loc
   ON big_table (akey)
   NOSORT
   NOLOGGING
   PARALLEL;
```

Cluster Index Example To create an index for the EMPLOYEE cluster, issue the following statement:

```
CREATE INDEX ic_emp ON CLUSTER employee;
```

No index columns are specified, because the index is automatically built on all the columns of the cluster key. For cluster indexes, all rows are indexed.

NULL Example Consider the following statement:

```
SELECT ename FROM emp WHERE comm IS NULL;
```

The above query does not use an index created on the COMM column unless it is a bitmap index.

Function-Based Index Example The following statements creates a function-based index on the EMP table based on an uppercase evaluation of the ENAME column:

```
CREATE INDEX emp_i ON emp (UPPER(ename));
```

To ensure that Oracle will use the index rather than performing a full table scan, be sure that the value of the function is not null in subsequent queries. For example, the statement

```
SELECT * FROM emp WHERE UPPER(ename) IS NOT NULL
   ORDER BY UPPER(ename);
```

is guaranteed to use the index, but without the *where_clause* Oracle may perform a full table scan.

Function-based Index on Type Method Example This example entails an object type RECTANGLE containing two number attributes: length and width. The AREA() method computes the area of the rectangle.

```
CREATE TYPE rectangle AS OBJECT
( length NUMBER,
  width NUMBER,
  MEMBER FUNCTION area RETURN NUMBER DETERMINISTIC
);

CREATE OR REPLACE TYPE BODY rectangle AS
  MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
   RETURN (length*width);
  END;
END;
```

Now, if you create a table RECTAB of type RECTANGLE, you can create a function-based index on the AREA() method as follows:

```
CREATE TABLE recttab OF rectangle;
CREATE INDEX area_idx ON recttab x (x.area());
```

You can use this index efficiently to evaluate a query of the form:

```
SELECT * FROM recttab x WHERE x.area() > 100;
```

Computing Statistics Example The following statement collects statistics on the nonpartitioned EMP_INDX index:

```
CREATE INDEX emp_indx ON emp(empno) COMPUTE STATISTICS;
```

The type of statistics collected depends on the type of index you are creating. For more information, refer to *Oracle8i Concepts*.

Partitioned Index Example The following statement creates a global prefixed index STOCK_IX on table STOCK_XACTIONS with two partitions, one for each half of the alphabet. The index partition names are system generated:

```
CREATE INDEX stock_ix ON stock_xactions
  (stock_symbol, stock_series)
   GLOBAL PARTITION BY RANGE (stock_symbol)
     (PARTITION VALUES LESS THAN ('N') TABLESPACE ts3,
      PARTITION VALUES LESS THAN (MAXVALUE) TABLESPACE ts4);
```

Index on Hash-Partitioned Table Example. This statement creates a local index on the ITEM column of the SALES table. The STORE IN clause immediately following LOCAL indicates that SALES is hash partitioned. Oracle will distribute the hash partitions between the TBS1 and TBS2 tablespaces:

```
CREATE INDEX sales_idx ON sales(item) LOCAL
   STORE IN (tbs1, tbs2);
```

Index on Composite-Partitioned Table Example. This statement creates a local index on the SALES table, which is composite-partitioned. The STORAGE clause specifies default storage attributes for the index. The STORE IN clause specifies one or more default tablespaces for the index subpartitions. However, this default is overridden for the four subpartitions of partition Q3_1997, because separate TABLESPACE is specified.

```
CREATE INDEX sales_idx ON sales(sale_date, item)
   STORAGE (INITIAL 1M, MAXEXTENTS UNLIMITED)
   LOCAL
   STORE IN (tbs1, tbs2, tbs3, tbs4, tbs5)
   (PARTITION q1_1997, PARTITION q2_1997,
    PARTITION q3_1997
      (SUBPARTITION q3_1997_s1 TABLESPACE ts2,
       SUBPARTITION q3_1997_s2 TABLESPACE ts4,
       SUBPARTITION q3_1997_s3 TABLESPACE ts6,
       SUBPARTITION q3_1997_s4 TABLESPACE ts8),
    PARTITION q4_1997,
    PARTITION q1_1998);
```

Bitmap Index Example To create a bitmap partitioned index on a table with four partitions, issue the following statement:

```
CREATE BITMAP INDEX partno_ix
   ON lineitem(partno)
   TABLESPACE ts1
   LOCAL (PARTITION quarter1 TABLESPACE ts2,
          PARTITION quarter2 STORAGE (INITIAL 10K NEXT 2K),
          PARTITION quarter3 TABLESPACE ts2,
          PARTITION quarter4);
```

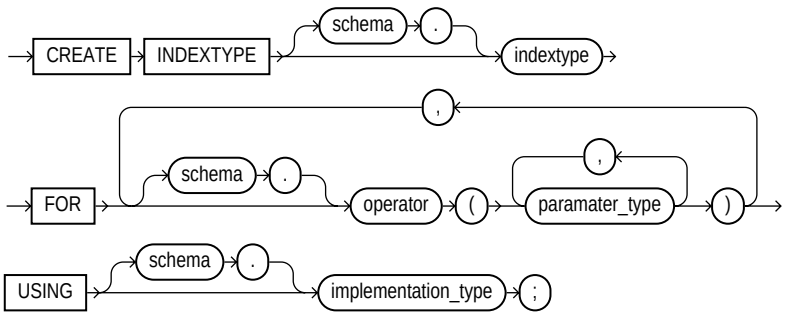
Nested Table Example In the following example, UNIQUE index UNIQ_PROJ_INDX is created on storage table NESTED_PROJECT_TABLE. Including pseudocolumn NESTED_TABLE_ID ensures distinct rows in nested table column PROJS_MANAGED:

```
CREATE TYPE proj_type AS OBJECT
   (proj_num NUMBER, proj_name VARCHAR2(20));
CREATE TYPE proj_table_type AS TABLE OF proj_type;
CREATE TABLE employee ( emp_num NUMBER, emp_name CHAR(31),
   projs_managed proj_table_type )
   NESTED TABLE projs_managed STORE AS nested_project_table;
```

```
CREATE UNIQUE INDEX uniq_proj_indx
   ON nested_project_table ( NESTED_TABLE_ID, proj_num);
```

CREATE INDEXTYPE

Syntax



Purpose

To create an **indextype**, which is an object that specifies the routines that manage a domain (application-specific) index. Indextypes reside in the same namespace as tables, views, and other schema objects. This statement binds the indextype name to an implementation type, which in turn specifies and refers to user-defined index functions and procedures that implement the indextype. For more information on implementing indextypes, see *Oracle8i Data Cartridge Developer's Guide* and *Oracle8i Concepts*.

Prerequisites

To create an indextype in your own schema, you must have the CREATE INDEXTYPE system privilege. To create an indextype in another schema, you must have CREATE ANY INDEXTYPE system privilege. In either case, you must have the EXECUTE object privilege on the implementation type and the supported operators.

An indextype supports one or more operators, so before creating an indextype, you should first design the operator or operators to be supported and provide functional implementation for those operators. For more information on operators, see "CREATE OPERATOR" on page 7-320.

Keywords and Parameters

schema	is the name of the schema in which the indextype resides. If you omit *schema*, Oracle creates the indextype in your own schema.	
indextype	is the name of the indextype to be created.	
FOR	specifies the list of operators supported by the indextype.	
	schema	is the schema containing the operator. If you omit *schema*, Oracle assumes the operator is in your own schema.
	operator	specifies the name of the operator supported by the indextype.
	parameter_type	lists the types of parameters to the operator.
	All the operators listed in this clause should be valid operators.	
USING	specifies the type that provides the implementation for the new indextype.	
	implementation_type	is the name of the type that implements the appropriate ODCI interface. ■ You must specify a valid type that implements the routines in the OCDI interface. ■ The implementation type must reside in the same schema as the indextype. For additional information on this interface, see *Oracle8i Data Cartridge Developer's Guide.*

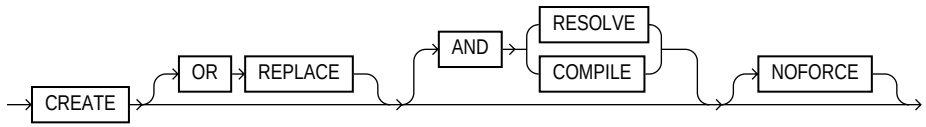
Example

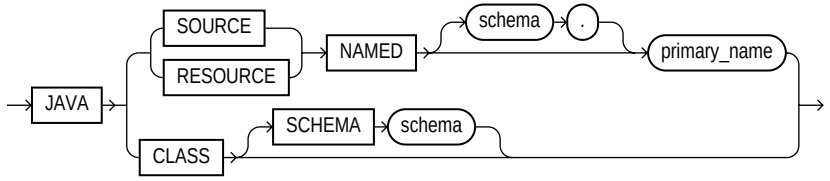
The following statement creates an indextype named TextIndexType and specifies the CONTAINS operator that is supported by the indextype and the TextIndexMethods type that implements the index interface:

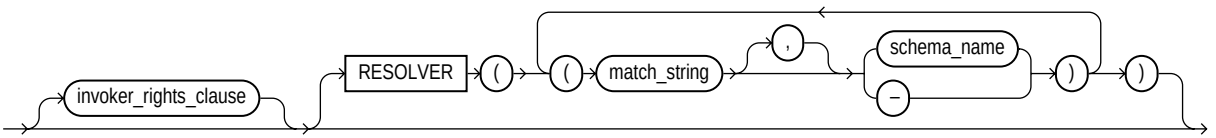
```
CREATE INDEXTYPE TextIndexType
   FOR contains (VARCHAR2, VARCHAR2)
   USING TextIndexMethods;
```

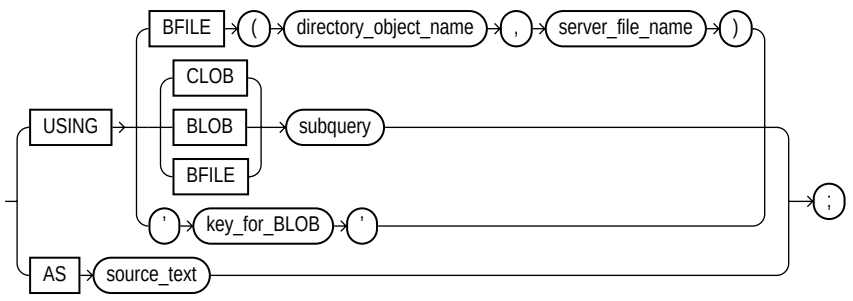
CREATE JAVA

Syntax

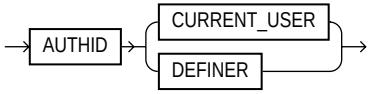








invoker_rights_clause::=



Purpose

To create a schema object containing a Java source, class, or resource. For information on the following topics, see these books:

- For Java concepts, see *Oracle8i Java Developer's Guide* .
- For Java stored procedures, see *Oracle8i Java Stored Procedures Developer's Guide*
- For SQLJ, see *Oracle8i SQLJ Developer's Guide and Reference* .
- For JDBC, see *Oracle8i JDBC Developer's Guide and Reference* .
- For CORBA and EJB, see *Oracle8i Enterprise JavaBeans and CORBA Developer's Guide*.

Prerequisites

To create or replace a schema object containing a Java source, class, or resource in your own schema, you must have CREATE PROCEDURE system privilege. To create such a schema object in another user's schema, you must have CREATE ANY PROCEDURE system privilege. To replace such a schema object in another user's schema, you must also have ALTER ANY PROCEDURE system privilege.

Keywords and Parameters

OR REPLACE re-creates the schema object containing the Java class, source, or resource if it already exists. Use this clause to change the definition of an existing object without dropping, re-creating, and regranting object privileges previously granted.

If you redefine a Java schema object and specify RESOLVE or COMPILE, Oracle recompiles or resolves the object. If the resolution or compilation is successful, Oracle does not invalidate classes that reference the Java schema object. For additional information, see "ALTER JAVA" on page 7-43.

Users who had previously been granted privileges on a redefined function can still access the function without being regranted the privileges.

RESOLVE | COMPILE are synonymous keywords. They specify that Oracle should attempt to resolve the Java schema object that is created if this statement succeeds.

- When applied to a class, resolution of referenced names to other class schema objects occurs.
- When applied to a source, source compilation occurs.

Restriction: You cannot specify this clause for a Java resource.

NOFORCE — rolls back the results of this CREATE command if you have specified either RESOLVE or COMPILE, and the resolution or compilation fails. If you do not specify this option, Oracle takes no action if the resolution or compilation fails (that is, the created schema object remains).

JAVA SOURCE — loads a Java source file.

JAVA CLASS — loads a Java class file.

JAVA RESOURCE — loads a Java resource file.

NAMED — is **required** for a Java source or resource.

- For a Java source, this clause specifies the name of the schema object in which the source code is held. A successful CREATE JAVA SOURCE statement will also create additional schema objects to hold each of the Java classes defined by the source.
- For a Java resource, this clause specifies the name of the schema object to hold the Java resource.

If you do not specify *schema*, Oracle creates the object in your own schema.

Restrictions:

- You cannot specify NAMED for a Java class.
- The *primary_name* cannot contain a database link.

SCHEMA *schema* — applies only to a Java class. This optional clause specifies the schema in which the object containing the Java file resides. If you do not specify SCHEMA and you do not specify NAMED (above), Oracle creates the object in your own schema.

invoker_rights_clause — specifies whether the methods of the class execute with the privileges and in the schema of the user who owns the class or with the privileges and in the schema of CURRENT_USER. For information on how CURRENT_USER is determined, see *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals.*

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type. For more information refer to *Oracle8i Java Stored Procedures Developer's Guide.*

AUTHID CURRENT_USER — specifies that the methods of the class execute with the privileges of CURRENT_USER. This clause is the default and creates an "invoker-rights class."

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the methods reside.

AUTHID DEFINER — specifies that the methods of the class execute with the privileges of the owner of the schema in which the class resides, and that external names resolve in the schema where the class resides.

RESOLVER — specifies a mapping of the fully qualified Java name to a Java schema object, where

- *match_string* is either a fully qualified Java name, a wildcard that can match such a Java name, or a wildcard that can match any name.
- *schema_name* designates a schema to be searched for the corresponding Java schema object.
- A dash (-) as an alternative to *schema_name* indicates that if *match_string* matches a valid Java name, Oracle can leave the schema unresolved. The resolution succeeds, but the name cannot be used at run time by the class.

This mapping is stored with the definition of the schema objects created in this command for use in later resolutions (either implicit or in explicit ALTER ... RESOLVE statements).

USING — determines a sequence of character (CLOB or BFILE) or binary (BLOB or BFILE) data for the Java class or resource. Oracle uses the sequence of characters to define one file for a Java class or resource, or one source file and one or more derived classes for a Java source.

BFILE	identifies a previously created file on the operating system (*directory_object_name*) and server file (*server_file_name*) containing the sequence. BFILE is usually interpreted as a character sequence by CREATE JAVA SOURCE and as a binary sequence by CREATE JAVA CLASS or CREATE JAVA RESOURCE.
CLOB / BLOB / BFILE *subquery*	supplies a query that selects a single row and column of the type specified (CLOB, BLOB, or BFILE). The value of the column makes up the sequence of characters.

key_for_BLOB	supplies the following implicit query:
	`SELECT LOB FROM CREATE$JAVA$LOB$TABLE`
	`WHERE NAME = 'key_for_BLOB';`
	Restriction: To use this case, the table `CREATE$JAVA$LOB$TABLE` must exist in the current schema and must have a column LOB of type BLOB and a column NAME of type VARCHAR2.

AS *source_text* — determines a sequence of characters for a Java or SQLJ source.

Examples

Java Class Example The following statement creates a schema object containing a Java class using the name found in a Java binary file:

```
CREATE JAVA CLASS USING BFILE (bfile_dir, 'Agent.class');
```

This example assumes the directory object `bfile_dir`, which points to the operating system directory containing the Java class `Agent.class`, already exists. In this example, the name of the class determines the name of the Java class schema object.

Java Source Example The following statement creates a Java source schema object:

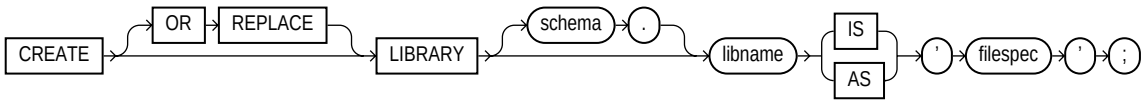
```
CREATE JAVA SOURCE NAMED "Hello" AS
   public class Hello (
      public static String hello() (
         return "Hello World";   ) );
```

Java Resource Example The following statement creates a Java resource schema object named APPTEXT from a BFILE:

```
CREATE JAVA RESOURCE NAMED "appText"
   USING BFILE (bfile_dir, 'textBundle.dat');
```

CREATE LIBRARY

Syntax



filespec: See "filespec" on page 7-490.

Purpose

To create a schema object associated with an operating-system shared library. The name of this schema object can then be used in the *call_spec* of CREATE FUNCTION or CREATE PROCEDURE statements, or when declaring a function or procedure in a package or type, so that SQL and PL/SQL can call to third-generation-language (3GL) functions and procedures. For more information on functions and procedures, see "CREATE FUNCTION" on page 7-266, "CREATE PROCEDURE" on page 7-333, and *PL/SQL User's Guide and Reference.*

Prerequisites

To create a library in your own schema, you must have the CREATE LIBRARY system privilege. To create a library in another user's schema, you must have the CREATE ANY LIBRARY system privilege. To use the procedures and functions stored in the library, you must have EXECUTE object privileges on the library.

The CREATE LIBRARY statement is valid only on platforms that support shared libraries and dynamic linking.

Keywords and Parameters

OR REPLACE	re-creates the library if it already exists. Use this clause to change the definition of an existing library without dropping, re-creating, and regranting schema object privileges granted on it. Users who had previously been granted privileges on a redefined library can still access the library without being regranted the privileges.
libname	is the name you with to create to represent this library when declaring a function or procedure with a *call_spec*.

'*filespec*'	is a string literal, enclosed in single quotes. This string should be the path or filename your operating system recognizes as naming the shared library. The 'filespec' is not interpreted during execution of the CREATE LIBRARY statement. The existence of the library file is not checked until an attempt is made to execute a routine from it.

Examples

The following statement creates library EXT_LIB:

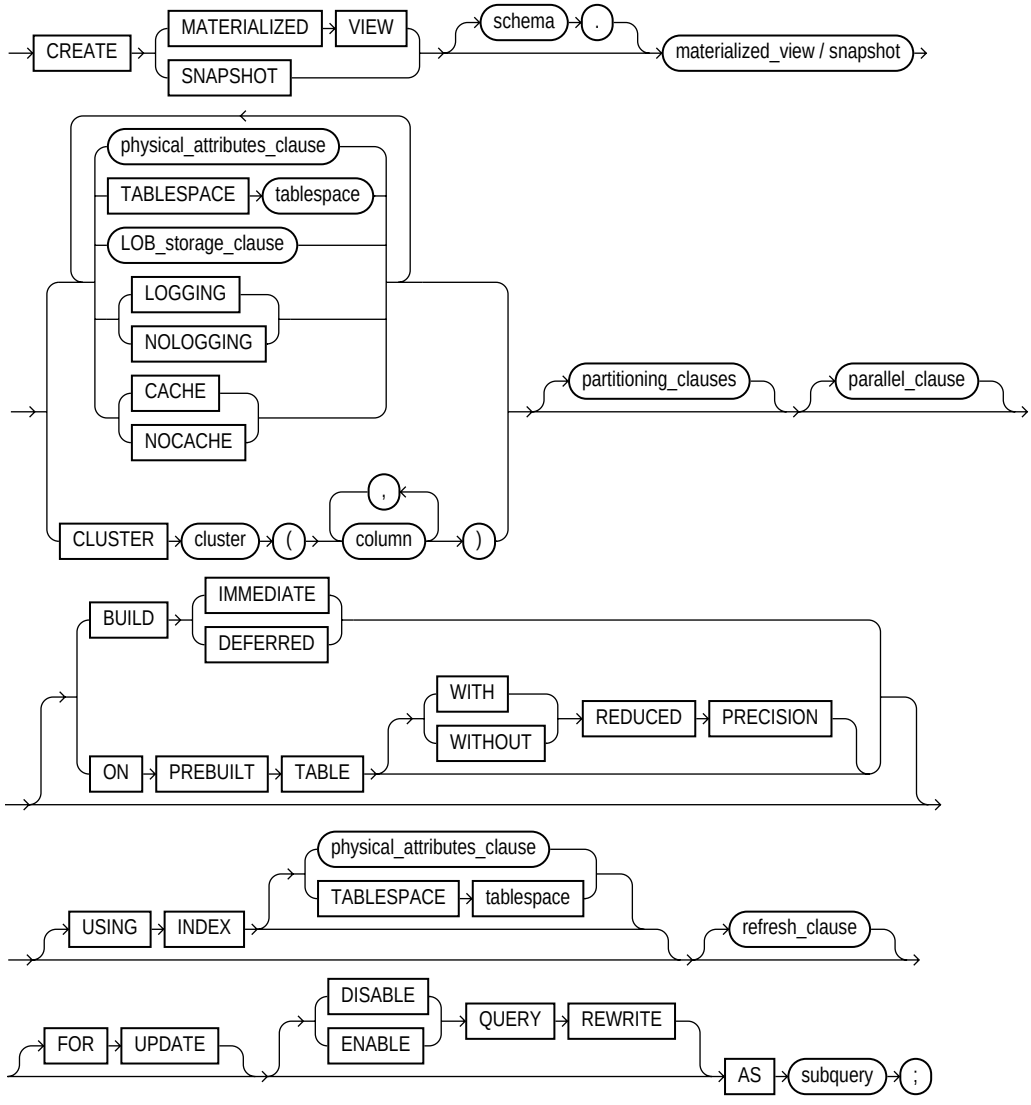
```
CREATE LIBRARY ext_lib AS '/OR/lib/ext_lib.so';
```

The following statement re-creates library EXT_LIB:

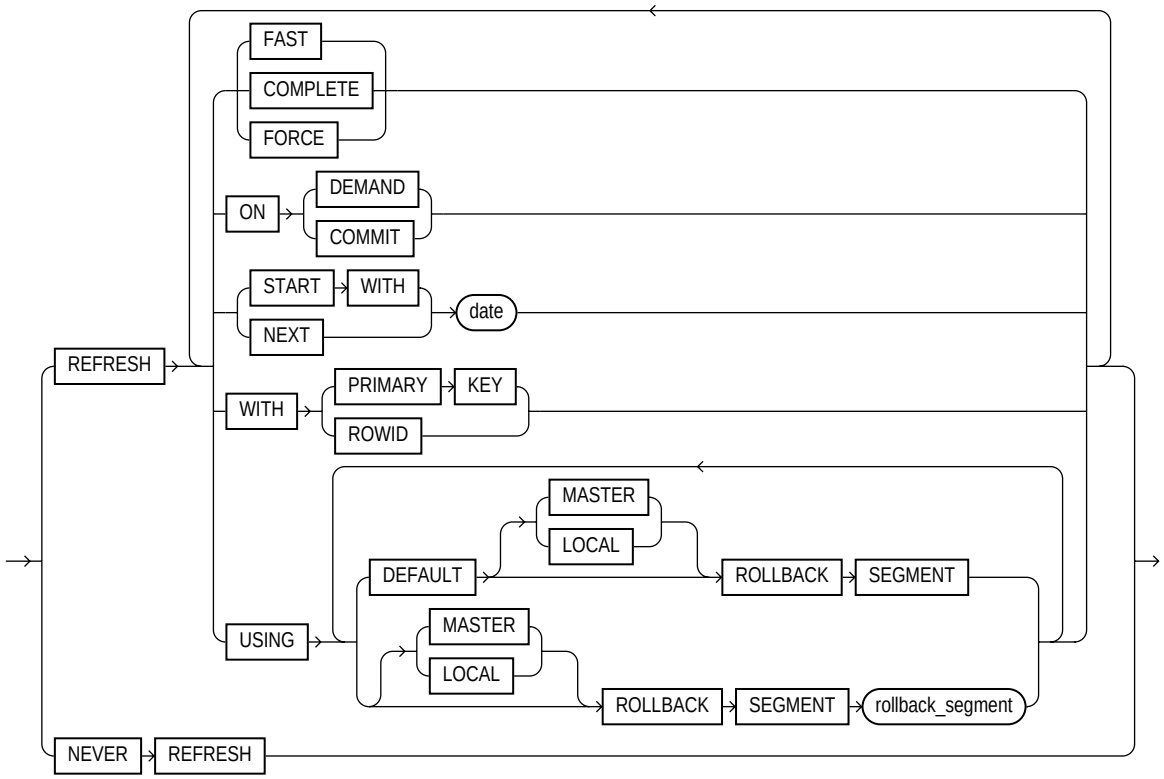
```
CREATE OR REPLACE LIBRARY ext_lib IS '/OR/newlib/ext_lib.so';
```

CREATE MATERIALIZED VIEW / SNAPSHOT

Syntax

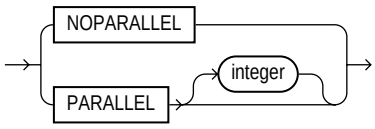


refresh_clause::=



physical_attributes_clause: See "CREATE TABLE" on page 7-359.

parallel_clause::=



subquery: See "SELECT and Subqueries" on page 7-541.

LOB_storage_clause: See "CREATE TABLE" on page 7-359.

partitioning_clauses: See "CREATE TABLE" on page 7-359.

Purpose

To create a **materialized view** or **snapshot**. The terms "snapshot" and "materialized view" are synonymous. Both refer to a table that contains the results of a query of one or more tables, each of which may be located on the same or on a remote database. The tables in the query are called *master tables* or *detail tables*. The databases containing the master tables are called the *master databases*.

For replication purposes, materialized views allow you to maintain copies of remote data on your local node. The copies are updatable with the Advanced Replication feature, read-only without this feature. You can select data from a materialized view as you would from a table or view. For more information on materialized views used to support replication, see *Oracle8i Replication*.

For data warehousing purposes, a materialized view definition can include an aggregation (SUM, COUNT(x), COUNT(*), COUNT(DISTINCT x), AVG, VARIANCE, STDDEV, MIN, and MAX) and any number of joins. Such materialized views can be used in query rewrite, an optimization technique that transforms a user request written in terms of master tables into a semantically equivalent request that includes one or more materialized view. In a data warehousing environment, all detail tables must be local.

Materialized views can take several forms. The various types of materialized views are discussed in *Oracle8i Tuning*.

Prerequisites

To create a materialized view in your own schema, you must have the CREATE SNAPSHOT or CREATE MATERIALIZED VIEW, CREATE TABLE, CREATE INDEX, and CREATE VIEW system privileges.

To create a materialized view in another user's schema, you must have the CREATE ANY SNAPSHOT or CREATE ANY MATERIALIZED VIEW system privilege.

To enable a materialized view for query rewrite:

- If all the master tables in the materialized view are in your schema, you must have the QUERY REWRITE privilege.
- If any of the master tables are in another schema, you must have the GLOBAL QUERY REWRITE privilege.
- If the materialized view is in another user's schema, **both you and the owner of that schema** must have the appropriate QUERY REWRITE privilege described in the preceding two items.

The schema that contains the materialized view must have sufficient quota in the target tablespace to store the materialized view's base table and index or have the UNLIMITED TABLESPACE system privilege.

To create and refresh a materialized view, both the creator and materialized view owner must be able to issue the defining query of the materialized view. This capability depends directly on the database link that the materialized view's defining query uses.

When you create a materialized view, Oracle creates one table, at least one index, and may create one view, all in the schema of the materialized view. Oracle uses these objects to maintain the materialized view's data. You must have the privileges necessary to create these objects. For information on these privileges, see "CREATE TABLE" on page 7-359, "CREATE VIEW" on page 7-430, and "CREATE INDEX" on page 7-273.

For complete information about the prerequisites that apply to creating materialized views for replication, see *Oracle8i Replication*. For complete information about the prerequisites that apply to creating materialized views for data warehousing, see *Oracle8i Tuning*.

Keywords and Parameters

schema	is the schema to contain the materialized view. If you omit *schema*, Oracle creates the materialized view in your schema.
materialized_view / snapshot	is the name of the materialized view to be created. Oracle generates names for the table and indexes used to maintain the materialized view by adding a prefix or suffix to the materialized view name. Oracle recommends that you limit your materialized view names to 19 bytes, so that the Oracle-generated names will be 30 bytes or less and will contain the entire materialized view name.
physical_ attributes_clause	establishes values for the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters (or, when used in the USING INDEX clause, for the INITRANS and MAXTRANS parameters only) and the storage parameters for the internal table Oracle uses to maintain the materialized view's data. For information on the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters, see "CREATE TABLE" on page 7-359. For information, about the *storage_clause*, see the "storage_clause" on page 7-575.
TABLESPACE	specifies the tablespace in which the materialized view is to be created. If you omit this clause, Oracle creates the materialized view in the default tablespace of the owner of the materialized view's schema.
LOB_storage_ clause	specifies the LOB storage characteristics. For detailed information about specifying the parameters of the *LOB_storage_clause*, see "CREATE TABLE" on page 7-359.

CLUSTER	creates the materialized view as part of the specified cluster. Since a clustered materialized view uses the cluster's space allocation, do not use the *physical_attributes_clause* or the TABLESPACE clause with the CLUSTER clause.
LOGGING | NOLOGGING	establishes the logging characteristics for the materialized view. For a description of logging characteristics, see "CREATE TABLE" on page 7-359.
CACHE | NOCACHE	determines where in the buffer cache Oracle stores blocks retrieved for the materialized view. For a description see "CREATE TABLE" on page 7-359.
partitioning_clauses	specifies that the materialized view is partitioned on specified ranges of values or on a hash function. Partitioning of materialized views is the same as partitioning tables, as described in "CREATE TABLE" on page 7-359.
parallel_clause	causes creation of the materialized view to be parallelized. For additional information, see the Notes to the *parallel_clause* of "CREATE TABLE" on page 7-359.
	NOPARALLEL — specifies serial execution. This is the default.
	PARALLEL — causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.
	PARALLEL *integer* — specifies the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.
BUILD	specifies when to populate the materialized view.
	IMMEDIATE — specifies that the materialized view is populated immediately. This is the default.
	DEFERRED — For **replication** purposes, this clause specifies that the materialized view will be populated at the next REFRESH operation. The first (deferred) refresh is always a complete refresh. Until then, the status of the materialized view is INVALID, so it cannot be used for query rewrite. For **data warehousing** purposes, this clause specifies that you will refresh the materialized view later manually using the DBMS_MVIEW.REFRESH procedure.
ON PREBUILT TABLE	lets you register an existing table to a preinitialized materialized view. The table must have the same name as the resulting materialized view. This is particularly useful for registering large materialized views in a data warehousing environment. The existing table object retains its identity as a table and is optionally maintained by the materialized view refresh mechanism to reflect changes made to the detail tables of *subquery*.

Restriction:

- At registration time, the table must reflect the materialization of a subquery.
- Each column alias in *subquery* must correspond to a column in *table_name*, and corresponding columns must have matching datatypes.
- If you specify this clause, you cannot specify a NOT NULL constraint for any column that is unmanaged (that is, not referenced in *subquery*) unless you also specify a default value for that column.

WITH REDUCED PRECISION — lets you authorize the loss of precision that will result if the precision of the table or materialized view columns do not exactly match the precision returned by *subquery*.

WITHOUT REDUCED PRECISION — requires that the precision of the table or materialized view columns match exactly the precision returned by *subquery*, or the create operation will fail. This is the default.

USING INDEX — specifies parameters for the indexes Oracle creates to maintain the materialized view. See *physical_attributes_clause*, above.

Restriction: You cannot specify the PCTUSED or PCTFREE parameters in this clause.

refresh_clause — specifies how and when Oracle automatically refreshes the materialized view. When a materialized view's master tables are modified, the data in a materialized view must be updated to ensure that the materialized view accurately reflects the data currently in its master table(s). This clause lets you schedule the times and specify the mode for Oracle to refresh the materialized view automatically.

Notes:

- This clause only sets the refresh options. For instructions on actually implementing the refresh, refer to *Oracle8i Replication* and *Oracle8i Tuning*.
- You can also refresh a materialized view immediately with the DBMS_MVIEW.REFRESH() procedure.

FAST — specifies a fast (incremental) refresh mode, which uses only the updated data stored in the materialized view log associated with the master or detail table. The appropriate log must exist for the fast refresh to succeed unless you use direct-path load.

Oracle can perform a fast refresh only if all of the following conditions are true:

- The materialized view conforms to the conditions defined in *Oracle8i Replication* (for replication use) and in *Oracle8i Tuning* (for data warehouse use).
- The materialized view's master table has a materialized view log or you used direct-load INSERT. (Oracle creates the direct loader log automatically. No user intervention is needed.)
- The necessary log was created before the materialized view was last refreshed or created.

Other restrictions may exist on the types of materialized views that you can fast refresh. For a complete explanation of when you can fast refresh a materialized view used for replication, see *Oracle8i Replication*. For a complete explanation of when you can fast refresh a materialized view used for data warehousing, see *Oracle8i Tuning*.

If you specify FAST for a materialized view with insufficient information to be incrementally refreshed, Oracle raises an error.

COMPLETE — specifies a complete refresh mode, or a refresh that reexecutes the materialized view's query. If you specify a complete refresh, Oracle performs a complete refresh regardless of whether a fast refresh is possible.

FORCE — specifies a fast refresh if one is possible or complete refresh if a fast refresh is not possible. Oracle decides whether a fast refresh is possible at refresh time.

If you omit FAST, COMPLETE, and FORCE, Oracle uses FORCE by default.

ON COMMIT — specifies that the refresh is to occur automatically when at the next COMMIT operation.

Restriction: This clause is supported only for materialized join views and materialized aggregate views. For further information, see *Oracle8i Replication* and *Oracle8i Tuning*.

ON DEMAND — specifies that materialized views will be refreshed on demand by calling one of the three DBMS_MVIEW procedures. For information on these procedures, see *Oracle8i Supplied Packages Reference*. The types of materialized views you can create by specifying refresh on demand are described in *Oracle8i Tuning*.

Alternatively, this clause specifies that a fast refresh will occur only if you add data using a direct-path method.

START WITH — specifies a date expression for the first automatic refresh time.

NEXT — specifies a date expression for calculating the interval between automatic refreshes.

Both the START WITH and NEXT values must evaluate to a time in the future. If you omit the START WITH value, Oracle determines the first automatic refresh time by evaluating the NEXT expression when you create the materialized view. If you specify a START WITH value but omit the NEXT value, Oracle refreshes the materialized view only once. If you omit both the START WITH and NEXT values, or if you omit the *refresh_clause* entirely, Oracle does not automatically refresh the materialized view.

WITH PRIMARY KEY — specifies that a primary-key materialized view is to be created. This is the default, and should be used in all cases except those described for WITH ROWID.

WITH ROWID — specifies that a rowid materialized view is to be created. Rowid materialized views provide compatibility with master tables in releases of Oracle prior to 8.0.

You can also use rowid materialized views to support selected materialized views that do not include all primary key columns. Rowid materialized views must be based on a single remote table and cannot contain any of the following:

- distinct or aggregate functions
- GROUP BY or CONNECT BY clauses
- subqueries
- joins
- set operations

Rowid materialized views cannot be fast refreshed after a master table reorganization.

USING ROLLBACK SEGMENT — specifies the remote rollback segment to be used during materialized view refresh, where *rollback_segment* is the name of the rollback segment to be used. (To change the local materialized view rollback segment, use the DBMS_REFRESH package, described in *Oracle8i Replication*.)

- DEFAULT specifies that Oracle will choose automatically which rollback segment to use. If you specify DEFAULT, you cannot specify *rollback_segment*. (Note: DEFAULT is most useful when modifying a materialized view, as described in "ALTER MATERIALIZED VIEW / SNAPSHOT" on page 7-45.)
- MASTER specifies the remote rollback segment to be used at the remote master for the individual materialized view.
- LOCAL specifies the remote rollback segment to be used for the local refresh group that contains the materialized view.

If you do not specify MASTER or LOCAL, Oracle uses LOCAL by default. If you do not specify *rollback_segment*, Oracle automatically chooses the rollback segment to be used.

The master rollback segment is stored on a per-materialized-view basis and is validated during materialized view creation and refresh. If the materialized view is complex, the master rollback segment, if specified, is ignored.

NEVER REFRESH — suppresses refresh of the materialized view. If you issue a REFRESH statement on the materialized view, Oracle returns an error.

FOR UPDATE — allows a subquery, primary key, or rowid materialized view to be updated. When used in conjunction with Advanced Replication, these updates will be propagated to the master. For more information, see *Oracle8i Replication*.

QUERY REWRITE — specifies whether the materialized view is eligible to be used for query rewrite.

ENABLE — enables the materialized view for query rewrite. For more information on query rewrite, see *Oracle8i Tuning*.

Note: Query rewrite is disabled by default, so you must specify this clause to make materialized views eligible for query rewrite.

Restrictions:

- You can enable query rewrite only if all user-defined functions in the materialized view are DETERMINISTIC. For more information, see "CREATE FUNCTION" on page 7-266.
- If you use bind variables in a query, the query will not be rewritten to use materialized views even if you enable query rewrite.
- You can enable query rewrite only if the statement contains only repeatable expressions. For example, you cannot include CURRENT_TIME or USER. For more information, see *Oracle8i Tuning*.

DISABLE — specifies that the materialized view is not eligible for use by query rewrite. However, a disabled materialized view can be refreshed.

AS *subquery* — specifies the materialized view query. When you create the materialized view, Oracle executes this query and places the results in the materialized view. This query is any valid SQL query. However, not all queries are fast refreshable, nor are all queries eligible for query rewrite.

Notes:

- Oracle does not execute the query immediately if you specify BUILD DEFERRED.
- Oracle recommends that you qualify each table and view in the FROM clause of the materialized view query with the schema containing it. For some additional caveats, see the AS *subquery* clause of "CREATE VIEW" on page 7-430. The restrictions described there for views apply to materialized views as well.

Restrictions:

- A materialized view query cannot select from tables or views owned by the user SYS.
- You cannot specify the ORDER BY clause in the subquery of a materialized view.
- Materialized views with a join or with multiple master tables and a GROUP BY clause cannot select from an index-organized table.
- Materialized views cannot contain columns of datatype LONG.
- If the subquery refers to a temporary table, you cannot create a materialized view log for this materialized view, nor can you specify the QUERY REWRITE clause of CREATE MATERIALIZED VIEW or ALTER MATERIALIZED VIEW.
- If the FROM list of the materialized view references another materialized view, you must control the refresh order of the materialized views manually. That is, you must refresh the materialized view depended upon and then the dependent materialized view in order to maintain integrity.

In addition, you should restrict the contents of *subquery* depending on what you hope to achieve with the materialized view, as follows:

If you want the materialized view to be eligible for fast refresh using a materialized view log, some restrictions apply. For more information on restrictions relating to replication, see *Oracle8i Replication*. For more information on restrictions relating to data warehousing, see *Oracle8i Tuning*.

If you are creating a materialized view enabled for query rewrite:

- The subquery cannot contain (either directly or through a view) references to ROWNUM, USER, SYSDATE, remote tables, sequences, or PL/SQL functions that write or read database or package state.
- The materialized view and detail tables of the materialized view must be local.

If you want to optimize query rewrite, the following additional guidelines apply:

- Do not specify a HAVING or CONNECT BY condition.
- Do not define any nested subqueries or inline views in the materialized view.
- If you specify a GROUP BY clause, it should not contain PL/SQL functions or expressions, and you should specify all of the GROUP BY columns in the SELECT list.
- All of the relations in the FROM list should be tables, and they should be distinct after synonym resolution.
- Specify outer joins for a complex materialized view, and list both sides of the outer join in the GROUP BY list.
- Ensure that each aggregated output expression uses one aggregate function (SUM, MIN, MAX, COUNT(x), COUNT(*), COUNT(DISTINCT x), AVG, VARIANCE, GROUPING, or STDDEV) with an expression that contains no explicit reference to any other grouping function.

Examples

Materialized Aggregate View Examples The following statement creates and populates a materialized view and specifies refresh mode and time:

```
CREATE MATERIALIZED VIEW mv1 REFRESH FAST ON COMMIT
   AS SELECT t.month, p.prod_name, SUM(f.sales) AS sum_sales
      FROM time t, product p, fact f
      WHERE f.curDate = t.curDate AND f.item = p.item
      GROUP BY t.month, p.prod_name
   BUILD IMMEDIATE;
```

The following statement creates and populates a materialized view SALES_BY_MONTH_BY_STATE. The materialized view will be populated with data as soon as the statement executes successfully, and subsequent refreshes will be accomplished by reexecuting the materialized view's query:

```
CREATE MATERIALIZED VIEW sales_by_month_by_state
   TABLESPACE my_ts PARALLEL (10)
   ENABLE QUERY REWRITE
   BUILD IMMEDIATE
   REFRESH COMPLETE
   AS SELECT t.month, g.state, SUM(sales) AS sum_sales
      FROM fact f, time t, geog g
      WHERE f.cur_date = t.cur_date AND f.city_id = g.city_id
      GROUP BY month, state;
```

The following statement creates a materialized view for an existing summary table, SALES_SUM_TABLE:

```
CREATE MATERIALIZED VIEW sales_sum_table
   ON PREBUILT TABLE
   ENABLE QUERY REWRITE
   AS SELECT t.month, g.state, SUM(sales)
      FROM fact f, time g, geog g
      WHERE f.cur_date = t.cur_date AND f.city_id = g.city_id
      GROUP BY month, state;
```

Materialized Join View Example The following statement creates a materialized join view MJV:

```
CREATE MATERIALIZED VIEW mjv
   REFRESH FAST
   START WITH 1-JUL-98
   NEXT SYSDATE +7 AS
      SELECT l.rowid as l_rid, l.pk, l.ofk, l.c1, l.c2,
         o.rowid as o_rid, o.pk, o.cfk, o.c1, o.c2,
         c.rowid as c_rid, c.pd, c.c1, c.c2
      FROM l, o, c
      WHERE l.ofk = o.pk(+) AND o.ofk = c.pk(+);
```

Subquery Materialized View Example The following statement creates a subquery materialized view based on the ORDERS table in the SALES schema at a remote database:

```
CREATE MATERIALIZED VIEW sales.orders FOR UPDATE
   AS SELECT * FROM sales.orders@dbs1.acme.com
   WHERE status = 'SHIPPABLE';
```

Primary Key Example The following statement creates primary-key materialized view HUMAN_GENOME:

```
CREATE SNAPSHOT human_genome
   REFRESH FAST START WITH SYSDATE NEXT  SYSDATE + 1/4096
   WITH PRIMARY KEY
   AS SELECT * FROM genome_catalog;
```

Rowid Example The following statement creates a rowid materialized view:

```
CREATE SNAPSHOT emp_data REFRESH WITH ROWID
    AS SELECT * FROM emp_table73;
```

Periodic Refresh Example The following statement creates the materialized view EMP_SF that contains the data from SCOTT's employee table in New York:

```
CREATE SNAPSHOT emp_sf
```

```
   PCTFREE 5 PCTUSED 60
   TABLESPACE users
   STORAGE (INITIAL 50K NEXT 50K)
   REFRESH FAST NEXT sysdate + 7
   AS SELECT * FROM scott.emp@ny;
```

The statement does not include a START WITH parameter, so Oracle determines the first automatic refresh time by evaluating the NEXT value using the current SYSDATE. Provided a materialized view log currently exists for the employee table in New York, Oracle performs a fast refresh of the materialized view every 7 days, beginning 7 days after the materialized view is created.

Because the materialized view conforms to the conditions for fast refresh, Oracle will perform a fast refresh. The above statement also establishes for the table storage characteristics that Oracle uses to maintain the materialized view.

Complete Refresh Example The following statement creates the materialized view ALL_EMPS that queries the employee tables in Dallas and Baltimore:

```
CREATE MATERIALIZED VIEW all_emps
   PCTFREE 5 PCTUSED 60
   TABLESPACE users
   STORAGE INITIAL 50K NEXT 50K
   USING INDEX STORAGE (INITIAL 25K NEXT 25K)
   REFRESH START WITH ROUND(SYSDATE + 1) + 11/24
   NEXT NEXT_DAY(TRUNC(SYSDATE, 'MONDAY') + 15/24
   AS SELECT * FROM fran.emp@dallas
         UNION
      SELECT * FROM marco.emp@balt;
```

Oracle automatically refreshes this materialized view tomorrow at 11:00 am and subsequently every Monday at 3:00 pm. ALL_EMPS contains a UNION, which is not supported for fast refresh, so Oracle automatically performs a complete refresh.

The above statement also establishes storage characteristics for both the table and the index that Oracle uses to maintain the materialized view:

- The first *storage_clause* establishes the sizes of the first and second extents of the table as 50 kilobytes each.
- The second *storage_clause* (appearing with the USING INDEX clause) establishes the sizes of the first and second extents of the index as 25 kilobytes each.

Rollback Segment Example The following statement creates materialized view SALE_EMP with rollback segment MASTER_SEG at the remote master and rollback segment SNAP_SEG for the local refresh group that contains the materialized view:

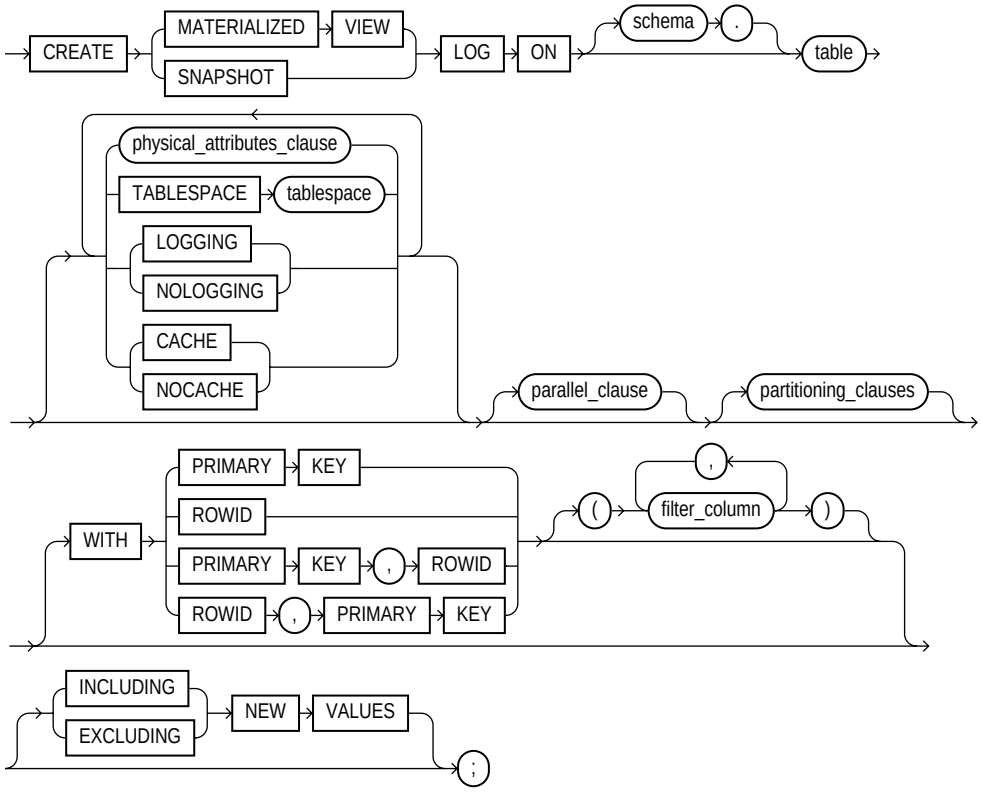
```
CREATE SNAPSHOT sales_emp
   REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 7
   USING MASTER ROLLBACK SEGMENT master_seg
   LOCAL ROLLBACK SEGMENT snap_seg
   AS SELECT * FROM bar;
```

The following statement is incorrect and generates an error because it specifies a segment name with a DEFAULT rollback segment:

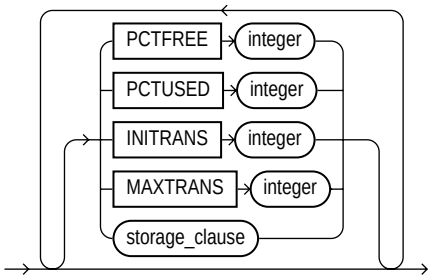
```
CREATE SNAPSHOT bogus
   REFRESH FAST START WITH SYSDATE NEXT SYSDATE + 7
   USING DEFAULT ROLLBACK SEGMENT snap_seg
   AS SELECT * FROM faux;
```

CREATE MATERIALIZED VIEW LOG / SNAPSHOT LOG

Syntax

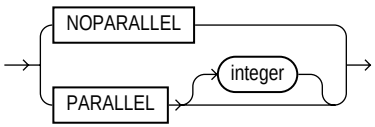


physical_attributes_clause::=



storage_clause: See "storage_clause" on page 7-575.

parallel_clause::=



partitioning_clauses: See "CREATE TABLE" on page 7-359.

Purpose

To create a materialized view log. A **materialized view log** is a table associated with the master table of a materialized view. When changes are made to the master table's data, Oracle stores rows describing those changes in the materialized view log and then uses the materialized view log to refresh materialized views based on the master table. This process is called a *fast refresh*. Without a materialized view log, Oracle must reexecute the materialized view query to refresh the materialized view. This process is called a *complete refresh*. Usually, a fast refresh takes less time than a complete refresh.

A materialized view log is located in the master database in the same schema as the master table. You need only a single materialized view log for a master table. Oracle can use this materialized view log to perform fast refreshes for all fast-refreshable materialized views based on the master table. For more information on materialized views, including how Oracle refreshes materialized views, see "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 7-300, *Oracle8i Tuning*, and *Oracle8i Replication*.

To fast refresh a materialized join view (a materialized view containing a join), you must create a materialized view log for each of its base tables. For more information on materialized views, see "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 7-300 and *Oracle8i Concepts.*

For information on modifying a materialized view log, see "ALTER MATERIALIZED VIEW LOG / SNAPSHOT LOG" on page 7-54. For information on dropping a materialized view log, see "DROP MATERIALIZED VIEW LOG / SNAPSHOT LOG" on page 7-461. Some types of materialized views are refreshed using a direct loader log. For information on using direct loader logs, see *Oracle8i Concepts.*

Prerequisites

The privileges required to create a materialized view log directly relate to the privileges necessary to create the underlying objects associated with a materialized view log.

- If you own the master table, you can create an associated materialized view log if you have the CREATE TABLE privilege.
- If you are creating a materialized view log for a table in another user's schema, you must have the CREATE ANY TABLE and COMMENT ANY TABLE privileges, as well as either the SELECT privilege for the master table or SELECT ANY TABLE.

In either case, the owner of the materialized view log must have sufficient quota in the tablespace intended to hold the materialized view log.

For detailed information about the prerequisites for creating a materialized view log, see *Oracle8i Replication*.

Keywords and Parameters

schema	is the schema containing the materialized view log's master table. If you omit *schema*, Oracle assumes the master table is contained in your own schema. Oracle creates the materialized view log in the schema of its master table. You cannot create a materialized view log for a table in the schema of the user SYS.
table	is the name of the master table for which the materialized view log is to be created. You cannot create a materialized view log for a view.
physical_ attributes_clause	establishes values for physical and storage characteristics for the materialized view log. See the descriptions of these parameters in "CREATE TABLE" on page 7-359 and "storage_ clause" on page 7-575.

TABLESPACE	specifies the tablespace in which the materialized view log is to be created. If you omit this clause, Oracle creates the materialized view log in the default tablespace the owner of the materialized view log's schema.
LOGGING | NOLOGGING	establishes the logging characteristics for the materialized view log. For a description of logging characteristics, see "CREATE TABLE" on page 7-359.
CACHE | NOCACHE	determines where in the buffer cache Oracle stores blocks retrieved for the materialized view log. For a description see "CREATE TABLE" on page 7-359.
parallel_clause	causes creation of the materialized view log to be parallelized. For additional information, see the Notes to the *parallel_clause* of "CREATE TABLE" on page 7-359.
	NOPARALLEL — specifies serial execution. This is the default.
	PARALLEL — causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.
	PARALLEL *integer* — specifies the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.
partitioning_ clauses	specifies that the materialized view log is partitioned on specified ranges of values or on a hash function. Partitioning of materialized view logs is the same as partitioning tables, as described in "CREATE TABLE" on page 7-359.
WITH	specifies whether the materialized view log should record the primary key, rowid, or both primary key and rowid when rows in the master are updated. This clause also specifies whether the materialized view log records filter columns, which are non-primary-key columns referenced by subquery materialized views.
	PRIMARY KEY — specifies that the primary key of all rows updated should be recorded in the materialized view log. The primary key of updated rows in the master table must be recorded in the materialized view log.
	ROWID — specifies that the rowid of all rows updated should be recorded in the materialized view log. The rowid must be recorded in the materialized view log.
	filter_column — is a comma-separated list that specifies the list of filter columns to be recorded in the materialized view log. For fast-refreshable primary-key materialized views defined with subqueries, all filter columns referenced by the defining subquery must be recorded in the materialized view log.
	Oracle records the primary key of all rows updated in the master by default.
NEW VALUES	specifies whether Oracle saves both old and new values in the materialized view log.

INCLUDING	saves old as well as new values in the log. If you are creating a log for a materialized aggregate view with only one master table, and if you want the materialized view to be eligible for fast refresh, you must specify INCLUDING.
EXCLUDING	saves only new values in the log. This is the default. To save overhead, use this clause for materialized join views and for materialized aggregate views with more than one master table. Such views do not require the old values.

Examples

Primary Key Examples The following statement creates a materialized view log on an employee table that records only primary key values:

```
CREATE SNAPSHOT LOG ON emp WITH PRIMARY KEY;
```

Oracle can use this materialized view log to perform a fast refresh on any simple primary key materialized view subsequently created on the EMP table.

The following statement also creates a materialized view log that record only the primary keys of updated rows:

```
CREATE SNAPSHOT LOG ON emp
   PCTFREE 5
   TABLESPACE users
   STORAGE (INITIAL 10K NEXT 10K);
```

ROWID Example The following statement creates a materialized view log that records both primary keys and rowids of updated rows:

```
CREATE SNAPSHOT LOG ON sales WITH ROWID, PRIMARY KEY;
```

Filter Column Example The following statement creates a materialized view log that records primary keys and updates to the filter column ZIP:

```
CREATE SNAPSHOT LOG ON address WITH (zip);
```

NEW VALUES Example The following example creates a master table, then creates a materialized view log that specifies INCLUDING NEW VALUES:

```
CREATE TABLE agg
   (u NUMBER, a NUMBER, b NUMBER, c NUMBER, d NUMBER);

CREATE MATERIALIZED VIEW LOG ON agg
   WITH ROWID (u,a,b,c,d)
```

```
   INCLUDING NEW VALUES;
```

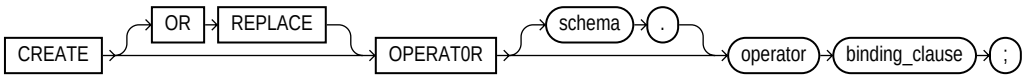
You could create the following materialized aggregate view to use the AGG log:

```
CREATE MATERIALIZED VIEW sn0
   REFRESH FAST ON COMMIT
   AS SELECT SUM(b+c), COUNT(*), a, d, COUNT(b+c)
         FROM agg
         GROUP BY a,d;
```

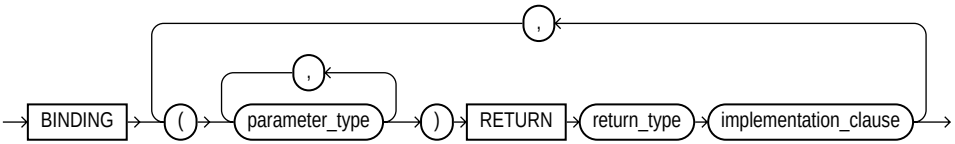
This materialized view is eligible for fast refresh because the log it uses includes both old and new values.

CREATE OPERATOR

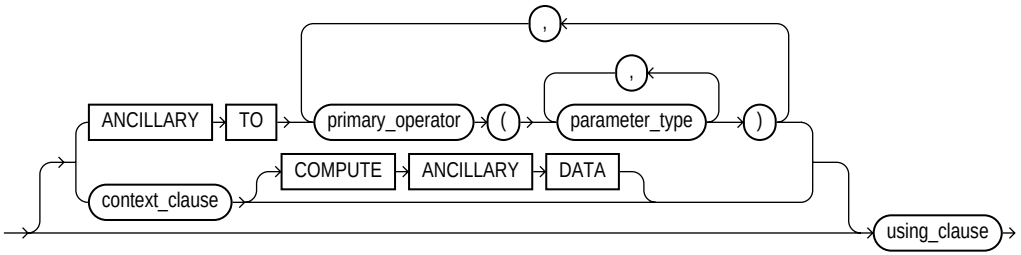
Syntax



binding_clause::=

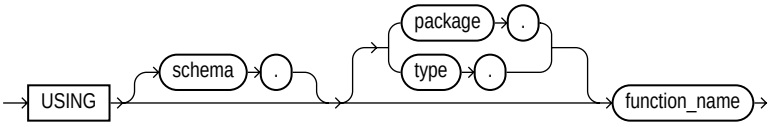


implementation_clause::=



context_clause::=

using_clause::=



Purpose

To create a new operator and define its bindings.

Operators can be referenced by indextypes and by DML and query SQL statements. The operators, in turn, reference functions, packages, types, and other user-defined objects. For a discussion of these dependencies, and of operators in general, see *Oracle8i Data Cartridge Developer's Guide* and *Oracle8i Concepts*.

Prerequisites

To create an operator in your own schema, you must have CREATE OPERATOR system privilege. To create an operator in another schema, you must have the CREATE ANY OPERATOR system privilege. In either case, you must also have EXECUTE privilege on the functions and operators referenced.

Keywords and Parameters

OR REPLACE — replaces the definition of the operator schema object.

Restriction: You can replace the definition only if the operator has no dependent objects (for example, indextypes supporting the operator).

schema — is the schema containing the operator. If you omit *schema*, Oracle assumes the operator is in your own schema.

operator — is the name of the operator to be created.

binding_clause — specifies one or more parameter datatypes (*parameter_type*) for binding the operator to a function. The signature of each binding (that is, the sequence of the datatypes of the arguments to the corresponding function) must be unique according to the rules of overloading. For more information about overloading, see *PL/SQL User's Guide and Reference*.

The *parameter_type* can itself be an object type. If it is, you can optionally qualify it with its schema.

Restriction: You cannot specify a *parameter_type* of REF, LONG, or LONG RAW.

RETURN — specifies the return datatype (*return_type*) for the binding.

The *return_type* can itself be an object type. If so, you can optionally qualify it with its schema.

Restriction: You cannot specify a *return_type* of REF, LONG, or LONG RAW.

implementation_clause

ANCILLARY TO *primary_operator* — specifies that the operator binding is ancillary to the specified primary operator binding (*primary_operator*). If you specify this clause, do not specify a previous binding with just one number parameter.

	context_clause	specifies the name of the implementation type used by the function as scan context.
	COMPUTE ANCILLARY DATA	specifies that the operator binding computes ancillary data.
using_clause	specifies the function that provides the implementation for the binding.	
	function_name	is the name of the function. The function can be a standalone function, packaged function, type method, or a synonym for any of these.

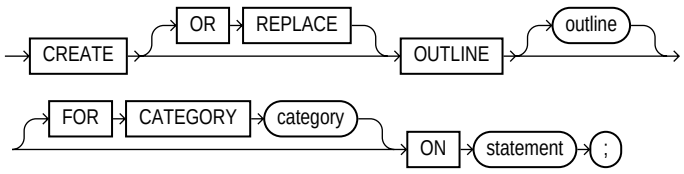
Example

This example creates an operator called MERGE in the SCOTT schema with two bindings. The first binding is for merging two VARCHAR2 values and returning a VARCHAR2 result. The second binding is for merging two geometries into a single geometry. The corresponding functional implementations for the bindings are also specified.

```
CREATE OPERATOR scott.merge
BINDING (varchar2, varchar2) RETURN varchar2
         USING text.merge,
   (spatial.geo, spatial.geo) RETURN spatial.geo
         USING spatial.merge;
```

CREATE OUTLINE

Syntax



Purpose

To create a **stored outline**, which is a set of attributes used by the optimizer to generate an execution plan. You can then instruct the optimizer to use a set of outlines to influence the generation of execution plans whenever a particular SQL statement is issued, regardless of changes in factors that can affect optimization. (To modify an outline so that it takes into account changes in these factors, see "ALTER OUTLINE" on page 7-58.)

You enable or disable the use of stored outlines dynamically for an individual session or for the system. See "ALTER SESSION" on page 7-78 and "ALTER SYSTEM" on page 7-95.

For more information on outlines, see also *Oracle8i Tuning*.

Prerequisites

To create an outline, you must have the CREATE ANY OUTLINE system privilege.

Keywords and Parameters

OR REPLACE	replaces an existing outline with a new outline of the same name.
outline	is the unique name to be assigned to the stored outline. If you do not specify *outline*, the system generates an outline name.
FOR CATEGORY *category*	specifies an optional name used to group stored outlines. For example, you could specify a category of outlines for end-of-week use and another for end-of-quarter use. If you do not specify *category*, the outline is stored in the DEFAULT category.
ON *statement*	is the SQL statement for which Oracle will create an outline when the statement is compiled. You can specify any one of the following statements:

- SELECT
- DELETE
- UPDATE
- INSERT ... SELECT
- CREATE TABLE ... AS SELECT

Note: You can specify multiple outlines for a single statement, but each outline for the same statement must be in a different category.

Example

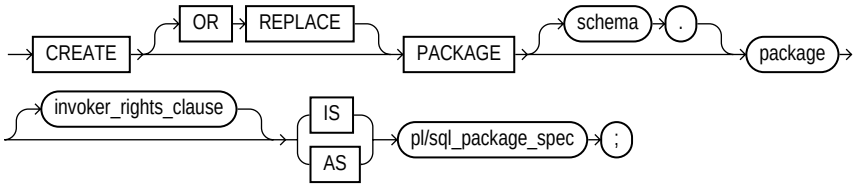
The following statement creates a stored outline by compiling the ON statement. The outline is called SALARIES and is stored in the category SPECIAL.

```
CREATE OUTLINE salaries FOR CATEGORY special
   ON SELECT ename, sal FROM emp;
```

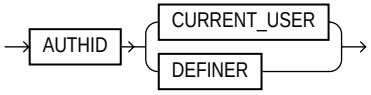
When this same SELECT statement is subsequently compiled, if the USE_STORED_OUTLINES parameter is set to SPECIAL, Oracle generates the same execution plan as was generated when the outline SALARIES was created.

CREATE PACKAGE

Syntax



invoker_rights_clause::=



Purpose

To create the specification for a stored package. A **package** is an encapsulated collection of related procedures, functions, and other program objects stored together in the database. The **specification** declares these objects.

For information on creating standalone functions and procedures, see "CREATE FUNCTION" on page 7-266 and "CREATE PROCEDURE" on page 7-333. For information on modifying a package, see "ALTER PACKAGE" on page 7-59. For information on dropping a package, see "DROP PACKAGE" on page 7-465.

For detailed discussions of packages and how to use them, see *Oracle8i Application Developer's Guide - Fundamentals* and *Oracle8i Supplied Packages Reference*.

Prerequisites

Before a package can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

To create a package in your own schema, you must have CREATE PROCEDURE system privilege. To create a package in another user's schema, you must have CREATE ANY PROCEDURE system privilege.

To embed a CREATE PACKAGE statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

For more information, see *PL/SQL User's Guide and Reference*.

Keywords and Parameters

OR REPLACE — re-creates the package specification if it already exists. Use this clause to change the specification of an existing package without dropping, re-creating, and regranting object privileges previously granted on the package. If you change a package specification, Oracle recompiles it. For information on recompiling package specifications, see "ALTER PACKAGE" on page 7-59.

Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.

If any function-based indexes depend on the package, Oracle marks the indexes DISABLED.

schema — is the schema to contain the package. If you omit *schema*, Oracle creates the package in your own schema.

package — is the name of the package to be created.

If creating the package results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SHOW ERRORS command.

invoker_rights_clause — lets you specify whether the functions and procedures in the package execute with the privileges and in the schema of the user who owns it or with the privileges and in the schema of CURRENT_USER. This specification applies to the corresponding package body as well. (For information on how CURRENT_USER is determined, see *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals*.)

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the package. For more information refer to *PL/SQL User's Guide and Reference*.

AUTHID CURRENT_USER — specifies that the package executes with the privileges of CURRENT_USER. This clause creates an "invoker-rights package."

This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the package resides.

AUTHID DEFINER — specifies that the package executes with the privileges of the owner of the schema in which the package resides and that external names resolve in the schema where the package resides. This is the default

pl/sql_package_spec	is the package specification, which can contain type definitions, cursor declarations, variable declarations, constant declarations, exception declarations, PL/SQL subprogram specifications, and call specifications (declarations of a C or Java routine expressed in PL/SQL). For a list of restrictions on user-defined functions in a package, see "Restrictions on User-Defined Functions" on page 7-268. For more information on PL/SQL package program units, see *PL/SQL User's Guide and Reference*. For information on Oracle supplied packages, see *Oracle8i Supplied Packages Reference*.

Example

The following SQL statement creates the specification of the EMP_MGMT package:

```
CREATE PACKAGE emp_mgmt AS
   FUNCTION hire(ename VARCHAR2, job VARCHAR2, mgr NUMBER,
                 sal NUMBER, comm NUMBER, deptno NUMBER)
      RETURN NUMBER;
   FUNCTION create_dept(dname VARCHAR2, loc VARCHAR2)
      RETURN NUMBER;
   PROCEDURE remove_emp(empno NUMBER);
   PROCEDURE remove_dept(deptno NUMBER);
   PROCEDURE increase_sal(empno NUMBER, sal_incr NUMBER);
   PROCEDURE increase_comm(empno NUMBER, comm_incr NUMBER);
      no_comm EXCEPTION;
      no_sal EXCEPTION;
END emp_mgmt;
```

The specification for the EMP_MGMT package declares the following public program objects:

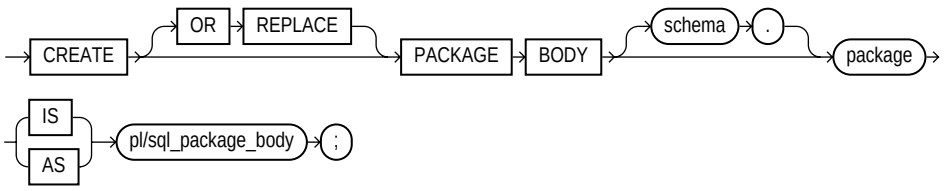
- the functions HIRE and CREATE_DEPT
- the procedures REMOVE_EMP, REMOVE_DEPT, INCREASE_SAL, and INCREASE_COMM
- the exceptions NO_COMM and NO_SAL

All of these objects are available to users who have access to the package. After creating the package, you can develop applications that call any of the package's public procedures or functions or raise any of the package's public exceptions.

Before you can call this package's procedures and functions, you must define these procedures and functions in the package body. For an example of a CREATE PACKAGE BODY statement that creates the body of the EMP_MGMT package, see "CREATE PACKAGE BODY" on page 7-328.

CREATE PACKAGE BODY

Syntax



Purpose

To create the body of a stored package. A **package** is an encapsulated collection of related procedures, stored functions, and other program objects stored together in the database. The **body** defines these objects. For information on creating standalone functions and procedures, see "CREATE FUNCTION" on page 7-266 and "CREATE PROCEDURE" on page 7-333.

Packages are an alternative to creating procedures and functions as standalone schema objects. For a discussion of packages, including how to create packages, see "CREATE PACKAGE" on page 7-325. For some illustrations, see "Examples" on page 7-329.

For information on modifying a package, see "ALTER PACKAGE" on page 7-59. For information on removing a package from the database, see "DROP PACKAGE" on page 7-465.

Prerequisites

Before a package can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

To create a package in your own schema, you must have CREATE PROCEDURE system privilege. To create a package in another user's schema, you must have CREATE ANY PROCEDURE system privilege.

To embed a CREATE PACKAGE BODY statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

For more information, see *PL/SQL User's Guide and Reference*.

Keywords and Parameters

OR REPLACE	re-creates the package body if it already exists. Use this clause to change the body of an existing package without dropping, re-creating, and regranting object privileges previously granted on it. If you change a package body, Oracle recompiles it. For information on recompiling package bodies, see "ALTER PACKAGE" on page 7-59. Users who had previously been granted privileges on a redefined package can still access the package without being regranted the privileges.
schema	is the schema to contain the package. If you omit *schema*, Oracle creates the package in your current schema.
package	is the name of the package to be created.
pl/sql_package_body	is the package body, which can contain PL/SQL subprogram bodies or call specifications (declarations of a C or Java routine expressed in PL/SQL). For a list of restrictions on user-defined functions in a package, see "Restrictions on User-Defined Functions" on page 7-268. For more information on writing a PL/SQL or C package program units, see *Oracle8i Application Developer's Guide - Fundamentals*. For information on JAVA package program units, see *Oracle8i Java Stored Procedures Developer's Guide*.

Examples

This SQL statement creates the body of the EMP_MGMT package:

```
CREATE PACKAGE BODY emp_mgmt AS
   tot_emps  NUMBER;
   tot_depts NUMBER;

FUNCTION hire
   (ename VARCHAR2,
    job VARCHAR2,
    mgr NUMBER,
    sal NUMBER,
    comm NUMBER,
    deptno NUMBER)

RETURN NUMBER IS
   new_empno NUMBER(4);
BEGIN
   SELECT empseq.NEXTVAL
      INTO new_empno
```

```
      FROM DUAL;
   INSERT INTO emp
      VALUES (new_empno, ename, job, mgr, sal, comm, deptno,
         tot_emps := tot_emps + 1;
   RETURN(new_empno);
END;

FUNCTION create_dept(dname VARCHAR2, loc VARCHAR2)
   RETURN NUMBER IS
      new_deptno NUMBER(4);
   BEGIN
      SELECT deptseq.NEXTVAL
         INTO new_deptno
         FROM dual;
      INSERT INTO dept
         VALUES (new_deptno, dname, loc);
            tot_depts := tot_depts + 1;
      RETURN(new_deptno);
   END;

PROCEDURE remove_emp(empno NUMBER) IS
   BEGIN
      DELETE FROM emp
      WHERE emp.empno = remove_emp.empno;
         tot_emps := tot_emps - 1;
   END;

PROCEDURE remove_dept(deptno NUMBER) IS
   BEGIN
      DELETE FROM dept
      WHERE dept.deptno = remove_dept.deptno;
         tot_depts := tot_depts - 1;
      SELECT COUNT(*)
         INTO tot_emps
         FROM emp;
         /* In case Oracle deleted employees from the EMP table
         to enforce referential integrity constraints, reset
         the value of the variable TOT_EMPS to the total
         number of employees in the EMP table. */
   END;

PROCEDURE increase_sal(empno NUMBER, sal_incr NUMBER) IS
   curr_sal NUMBER(7,2);
   BEGIN
      SELECT sal
```

```
      INTO curr_sal
      FROM emp
      WHERE emp.empno = increase_sal.empno;
      IF curr_sal IS NULL
         THEN RAISE no_sal;
      ELSE
         UPDATE emp
         SET sal = sal + sal_incr
         WHERE empno = empno;
      END IF;
   END;

PROCEDURE increase_comm(empno NUMBER, comm_incr NUMBER) IS
      curr_comm NUMBER(7,2);
   BEGIN
      SELECT comm
      INTO curr_comm
      FROM emp
      WHERE emp.empno = increase_comm.empno
      IF curr_comm IS NULL
         THEN RAISE no_comm;
      ELSE
         UPDATE emp
         SET comm = comm + comm_incr;
      END IF;
   END;

END emp_mgmt;
```

This package body corresponds to the package specification in the example of the "CREATE PACKAGE" statement earlier in this chapter. The package body defines the public program objects declared in the package specification:

- the functions HIRE and CREATE_DEPT
- the procedures REMOVE_EMP, REMOVE_DEPT, INCREASE_SAL, and INCREASE_COMM

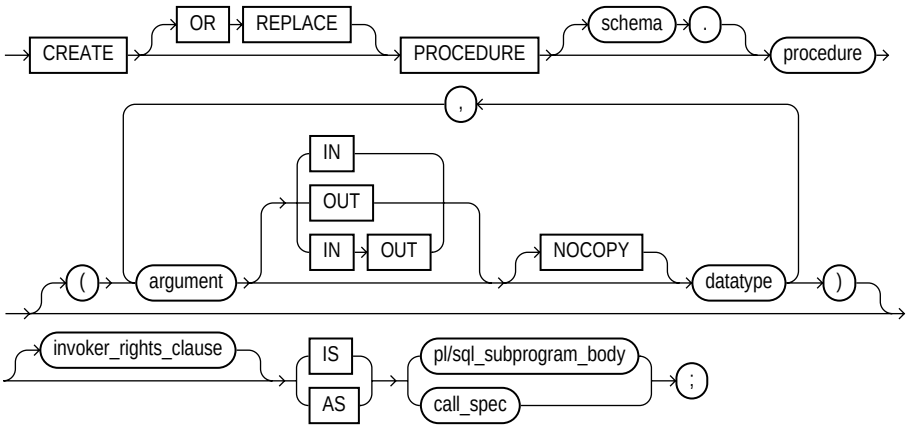
These objects are *declared* in the package specification, so they can be called by application programs, procedures, and functions outside the package. For example, if you have access to the package, you can create a procedure INCREASE_ALL_COMMS separate from the EMP_MGMT package that calls the INCREASE_COMM procedure.

These objects are *defined* in the package body, so you can change their definitions without causing Oracle to invalidate dependent schema objects. For example, if you subsequently change the definition of HIRE, Oracle need not recompile INCREASE_ALL_COMMS before executing it.

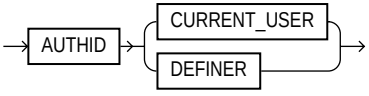
The package body in this example also declares private program objects, the variables TOT_EMPS and TOT_DEPTS. These objects are declared in the package body rather than the package specification, so they are accessible to other objects in the package, but they are not accessible outside the package. For example, you cannot develop an application that explicitly changes the value of the variable TOT_DEPTS. However, the function CREATE_DEPT is part of the package, so CREATE_DEPT can change the value of TOT_DEPTS.

CREATE PROCEDURE

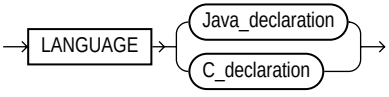
Syntax



invoker_rights_clause::=

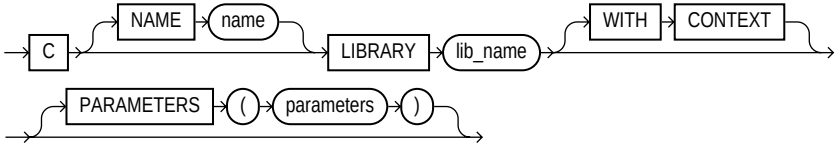


call_spec::=



Java_declaration::=

C_declaration::=



Purpose

To create a standalone stored procedure or a call specification.

A **procedure** is a group of PL/SQL statements that you can call by name. A **call specification ("call spec")** declares a Java method or a third-generation language (3GL) routine so that it can be called from SQL and PL/SQL. The call spec tells Oracle which Java method to invoke when a call is made. It also tells Oracle what type conversions to make for the arguments and return value.

Stored procedures offer advantages in the areas of development, integrity, security, performance, and memory allocation. For more information on stored procedures, including how to call stored procedures, see *Oracle8i Application Developer's Guide - Fundamentals*.

Stored procedures and stored functions are similar in many ways. For information specific to functions, see "CREATE FUNCTION" on page 7-266.

The CREATE PROCEDURE statement creates a procedure as a standalone schema object. You can also create a procedure as part of a package. For information on creating packages, see "CREATE PACKAGE" on page 7-325.

For information on modifying and dropping a standalone procedure, see "ALTER PROCEDURE" on page 7-62 and "DROP PROCEDURE" on page 7-467.

For more information about shared libraries, see "CREATE LIBRARY" on page 7-298. For more information about registering external procedures, see the *Oracle8i Application Developer's Guide - Fundamentals*.

Prerequisites

Before creating a procedure, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depends on your operating system.

To create a procedure in your own schema, you must have the CREATE PROCEDURE system privilege. To create a procedure in another user's schema, you must have CREATE ANY PROCEDURE system privilege. To replace a

procedure in another schema, you must have the ALTER ANY PROCEDURE system privilege.

To invoke a call spec, you may need additional privileges (for example, EXECUTE privileges on the C library for a C call spec). For more information on such prerequisites, refer to *PL/SQL User's Guide and Reference* or *Oracle8i Java Stored Procedures Developer's Guide.*

To embed a CREATE PROCEDURE statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

Keywords and Parameters

OR REPLACE	re-creates the procedure if it already exists. Use this clause to change the definition of an existing procedure without dropping, re-creating, and regranting object privileges previously granted on it. If you redefine a procedure, Oracle recompiles it. For information on recompiling procedures, see "ALTER PROCEDURE" on page 7-62. Users who had previously been granted privileges on a redefined procedure can still access the procedure without being regranted the privileges. If any function-based indexes depend on the package, Oracle marks the indexes DISABLED.
schema	is the schema to contain the procedure. If you omit *schema*, Oracle creates the procedure in your current schema.
procedure	is the name of the procedure to be created. If creating the procedure results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.
argument	is the name of an argument to the procedure. If the procedure does not accept arguments, you can omit the parentheses following the procedure name.
IN	specifies that you must specify a value for the argument when calling the procedure.
OUT	specifies that the procedure passes a value for this argument back to its calling environment after execution.
IN OUT	specifies that you must specify a value for the argument when calling the procedure and that the procedure passes a value back to its calling environment after execution. If you omit IN, OUT, and IN OUT, the argument defaults to IN.
NOCOPY	instructs Oracle to pass this argument as fast as possible. This clause can significantly enhance performance when passing a large value like a record, a PL/SQL table, or a varray to an OUT or IN OUT parameter. (IN parameter values are always passed NOCOPY.)

- When you specify NOCOPY, assignments made to a package variable may show immediately in this parameter (or assignments made to this parameter may show immediately in a package variable) if the package variable is passed as the actual assignment corresponding to this parameter.
- Similarly, changes made either to this parameter or to another parameter may be visible immediately through both names if the same variable is passed to both.
- If the procedure is exited with an unhandled exception, any assignment made to this parameter may be visible in the caller's variable.

These effects may or may not occur on any particular call. You should use NOCOPY only when these effects would not matter.

datatype is the datatype of the argument. An argument can have any datatype supported by PL/SQL.

Datatypes cannot specify length, precision, or scale. For example, VARCHAR2(10) is not valid, but VARCHAR2 is valid. Oracle derives the length, precision, and scale of an argument from the environment from which the procedure is called.

invoker_rights_clause lets you specify whether the procedure executes with the privileges and in the schema of the user who owns it or with the privileges and in the schema of CURRENT_USER. (For information on how CURRENT_USER is determined, see *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals.*)

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the procedure. For more information refer to *PL/SQL User's Guide and Reference*.

AUTHID CURRENT_USER	specifies that the procedure executes with the privileges of CURRENT_USER. This clause creates an "invoker-rights procedure." This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the procedure resides.
AUTHID DEFINER	specifies that the procedure executes with the privileges of the owner of the schema in which the procedure resides, and that external names resolve in the schema where the procedure resides. This is the default.

pl/sql_subprogram_body declares the procedure in a PL/SQL subprogram body. For more information on PL/SQL subprograms, see *Oracle8i Application Developer's Guide - Fundamentals*.

call_spec maps a Java or C method name, parameter types, and return type to their SQL counterparts.

- In *Java_declaration*, '*string*' identifies the Java implementation of the method. For more information, see *Oracle8i Java Stored Procedures Developer's Guide*.
- For an explanation of the parameters and semantics of *the C_declaration*, see *Oracle8i Application Developer's Guide - Fundamentals*.

AS EXTERNAL	is an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle Corporation recommends that you use the AS LANGUAGE C syntax.

Examples

The following statement creates the procedure CREDIT in the schema SAM:

```
CREATE PROCEDURE sam.credit (acc_no IN NUMBER, amount IN NUMBER) AS
   BEGIN
      UPDATE accounts
      SET balance = balance + amount
      WHERE account_id = acc_no;
   END;
```

The CREDIT procedure credits a specified bank account with a specified amount. When you call the procedure, you must specify the following arguments:

ACC_NO	is the number of the bank account to be credited. The argument's datatype is NUMBER.
AMOUNT	is the amount of the credit. The argument's datatype is NUMBER.

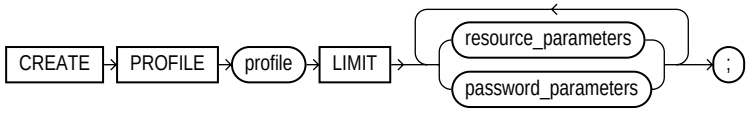
The procedure uses an UPDATE statement to increase the value in the BALANCE column of the ACCOUNTS table by the value of the argument AMOUNT for the account identified by the argument ACC_NO.

In the following example, external procedure C_FIND_ROOT expects a pointer as a parameter. Procedure FIND_ROOT passes the parameter by reference using the BY REF phrase:

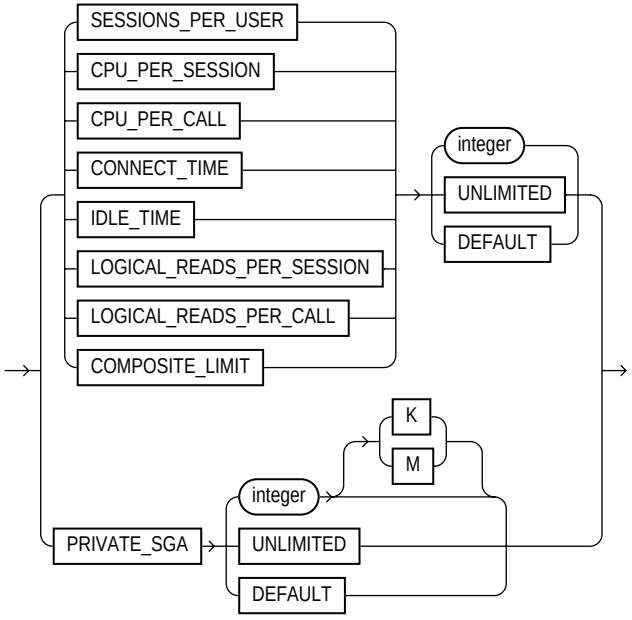
```
CREATE PROCEDURE find_root
   ( x IN REAL )
   IS LANGUAGE C
      NAME "c_find_root"
      LIBRARY c_utils
      PARAMETERS ( x BY REF );
```

CREATE PROFILE

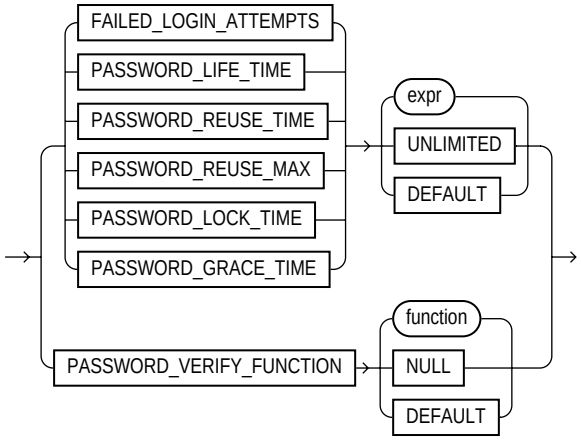
Syntax



resource_parameters::=



password_parameters::=



Purpose

To create a profile. A **profile** is a set of limits on database resources. If you assign the profile to a user, that user cannot exceed these limits.

Prerequisites

You must have CREATE PROFILE system privilege.

To specify resource limits for a user, you must:

- Enable resource limits dynamically with the ALTER SYSTEM statement (see "ALTER SYSTEM" on page 7-95) or with the initialization parameter RESOURCE_LIMIT. (This parameter does not apply to password resources. Password resources are always enabled.)
- Create a profile that defines the limits using the CREATE PROFILE statement.
- Assign the profile to the user using the CREATE USER or ALTER USER statement (see "CREATE USER" on page 7-425 and "ALTER USER" on page 7-179).

Keywords and Parameters

profile is the name of the profile to be created. Use profiles to limit the database resources available to a user for a single call or a single session.

Oracle enforces resource limits in the following ways:

- If a user exceeds the CONNECT_TIME or IDLE_TIME session resource limit, Oracle rolls back the current transaction and ends the session. When the user process next issues a call, Oracle returns an error.
- If a user attempts to perform an operation that exceeds the limit for other session resources, Oracle aborts the operation, rolls back the current statement, and immediately returns an error. The user can then commit or roll back the current transaction, and must then end the session.
- If a user attempts to perform an operation that exceeds the limit for a single call, Oracle aborts the operation, rolls back the current statement, and returns an error, leaving the current transaction intact.

Notes:

- You can use fractions of days for all parameters that limit time, with days as units. For example, 1 hour is 1/24 and 1 minute is 1/1440.
- You can specify resource limits for users regardless of whether the resource limits are enabled. However, Oracle does not enforce the limits until you enable them.

UNLIMITED When specified with a resource parameter, indicates that a user assigned this profile can use an unlimited amount of this resource. When specified with a password parameter, indicates that no limit has been set for the parameter.

DEFAULT omits a limit for this resource in this profile. A user assigned this profile is subject to the limit for this resource specified in the DEFAULT profile. The DEFAULT profile initially defines unlimited resources. You can change those limits with the ALTER PROFILE statement.

Any user who is not explicitly assigned a profile is subject to the limits defined in the DEFAULT profile. Also, if the profile that is explicitly assigned to a user omits limits for some resources or specifies DEFAULT for some limits, the user is subject to the limits on those resources defined by the DEFAULT profile.

resource_parameters

SESSIONS_PER_USER limits a user to *integer* concurrent sessions.

CPU_PER_SESSION limits the CPU time for a session, expressed in hundredth of seconds.

CPU_PER_CALL limits the CPU time for a call (a parse, execute, or fetch), expressed in hundredths of seconds.

CONNECT_TIME	limits the total elapsed time of a session, expressed in minutes.
IDLE_TIME	limits periods of continuous inactive time during a session, expressed in minutes. Long-running queries and other operations are not subject to this limit.
LOGICAL_READS_PER_SESSION	specifies the number of data blocks read in a session, including blocks read from memory and disk.
LOGICAL_READS_PER_CALL	specifies the number of data blocks read for a call to process a SQL statement (a parse, execute, or fetch).
PRIVATE_SGA	specifies the amount of private space a session can allocate in the shared pool of the system global area (SGA), expressed in bytes. Use K or M to specify this limit in kilobytes or megabytes. **Note:** This limit applies only if you are using multi-threaded server architecture. The private space for a session in the SGA includes private SQL and PL/SQL areas, but not shared SQL and PL/SQL areas.
COMPOSITE_LIMIT	specifies the total resources cost for a session, expressed in *service units*. Oracle calculates the total service units as a weighted sum of CPU_PER_SESSION, CONNECT_TIME, LOGICAL_READS_PER_SESSION, and PRIVATE_SGA. For information on how to specify the weight for each session resource, see "ALTER RESOURCE COST" on page 7-68.
password_parameters	For a detailed description and explanation of how to use password management and protection, see *Oracle8i Administrator's Guide*.
FAILED_LOGIN_ATTEMPTS	specifies the number of failed attempts to log in to the user account before the account is locked.
PASSWORD_LIFE_TIME	limits the number of days the same password can be used for authentication. The password expires if it is not changed within this period, and further connections are rejected.
PASSWORD_REUSE_TIME	specifies the number of days before which a password cannot be reused. If you set PASSWORD_REUSE_TIME to an integer value, then you must set PASSWORD_REUSE_MAX to UNLIMITED.
PASSWORD_REUSE_MAX	specifies the number of password changes required before the current password can be reused. If you set PASSWORD_REUSE_MAX to an integer value, then you must set PASSWORD_REUSE_TIME to UNLIMITED.
PASSWORD_LOCK_TIME	specifies the number of days an account will be locked after the specified number of consecutive failed login attempts.
PASSWORD_GRACE_TIME	specifies the number of days after the grace period begins during which a warning is issued and login is allowed. If the password is not changed during the grace period, the password expires.

PASSWORD_VERIFY_ FUNCTION	allows a PL/SQL password complexity verification script to be passed as an argument to the CREATE PROFILE statement. Oracle provides a default script, but you can create your own routine or use third-party software instead.	
	function	is the name of the password complexity verification routine.
	NULL	indicates that no password verification is performed.

Restrictions on password parameters:

- If PASSWORD_REUSE_TIME is set to an integer value, PASSWORD_REUSE_MAX must be set to UNLIMITED. If PASSWORD_REUSE_MAX is set to an integer value, PASSWORD_REUSE_TIME must be set to UNLIMITED.
- If both PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX are set to UNLIMITED, then Oracle uses neither of these password resources.
- If PASSWORD_REUSE_MAX is set to DEFAULT and PASSWORD_REUSE_TIME is set to UNLIMITED, then Oracle uses the PASSWORD_REUSE_MAX value defined in the DEFAULT profile.
- If PASSWORD_REUSE_TIME is set to DEFAULT and PASSWORD_REUSE_MAX is set to UNLIMITED, then Oracle uses the PASSWORD_REUSE_TIME value defined in the DEFAULT profile.
- If both PASSWORD_REUSE_TIME and PASSWORD_REUSE_MAX are set to DEFAULT, then Oracle uses whichever value is defined in the DEFAULT profile.

Examples

The following statement creates the profile SYSTEM_MANAGER:

```
CREATE PROFILE system_manager
   LIMIT SESSIONS_PER_USER    UNLIMITED
   CPU_PER_SESSION            UNLIMITED
   CPU_PER_CALL               3000
   CONNECT_TIME               45
   LOGICAL_READS_PER_SESSION  DEFAULT
   LOGICAL_READS_PER_CALL     1000
   PRIVATE SGA                15K
   COMPOSITE_LIMIT            5000000;
```

If you then assign the SYSTEM_MANAGER profile to a user, the user is subject to the following limits in subsequent sessions:

- The user can have any number of concurrent sessions.
- In a single session, the user can consume an unlimited amount of CPU time.
- A single call made by the user cannot consume more than 30 seconds of CPU time.

- A single session cannot last for more than 45 minutes.
- In a single session, the number of data blocks read from memory and disk is subject to the limit specified in the DEFAULT profile.
- A single call made by the user cannot read more than 1000 data blocks from memory and disk.
- A single session cannot allocate more than 15 kilobytes of memory in the SGA.
- In a single session, the total resource cost cannot exceed 5 million service units. The formula for calculating the total resource cost is specified by the ALTER RESOURCE COST statement.
- Since the SYSTEM_MANAGER profile omits a limit for IDLE_TIME and for password limits, the user is subject to the limits on these resources specified in the DEFAULT profile.

The following statement creates the profile PROF:

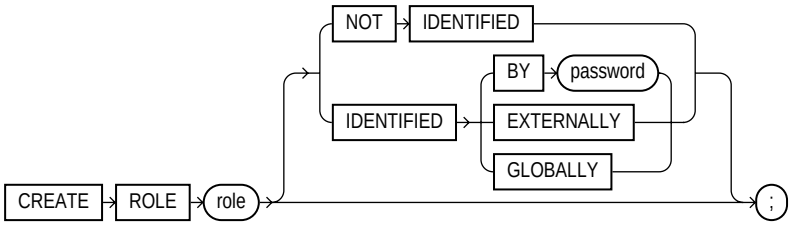
```
CREATE PROFILE prof
  LIMIT PASSWORD_REUSE_MAX DEFAULT
        PASSWORD_REUSE_TIME UNLIMITED;
```

The following statement creates profile MYPROFILE with password profile limits values set:

```
CREATE PROFILE myprofile LIMIT
   FAILED_LOGIN_ATTEMPTS 5
   PASSWORD_LIFE_TIME 60
   PASSWORD_REUSE_TIME 60
   PASSWORD_REUSE_MAX UNLIMITED
   PASSWORD_VERIFY_FUNCTION verify_function
   PASSWORD_LOCK_TIME 1/24
   PASSWORD_GRACE_TIME 10;
```

CREATE ROLE

Syntax



Purpose

To create a **role**, which is a set of privileges that can be granted to users or to other roles. You can use roles to administer database privileges. You can add privileges to a role and then grant the role to a user. The user can then enable the role and exercise the privileges granted by the role.

A role contains all privileges granted to the role and all privileges of other roles granted to it. A new role is initially empty. You add privileges to a role with the GRANT statement. For information on granting roles, see "GRANT system_ privileges_and_roles" on page 7-493. For information on enabling roles, see "ALTER USER" on page 7-179.

When you create a role that is NOT IDENTIFIED or is IDENTIFIED EXTERNALLY or BY *password*, Oracle grants you the role with ADMIN OPTION. However, when you create a role IDENTIFIED GLOBALLY, Oracle does not grant you the role.

For information on modifying a role, see"ALTER ROLE" on page 7-71. For information on removing a role from the database, see "DROP ROLE" on page 7-469. For information on enabling and disabling roles for the current session, see "SET ROLE" on page 7-570. For a detailed description and explanation of using global roles, see *Oracle8i Distributed Database Systems.*

Prerequisites

You must have CREATE ROLE system privilege.

Keywords and Parameters

role is the name of the role to be created. Oracle recommends that the role contain at least one single-byte character regardless of whether the database character set also contains multibyte characters.

Some roles are defined by SQL scripts provided on your distribution media. For a list of these predefined roles, see "GRANT system_privileges_and_roles" on page 7-493.

NOT IDENTIFIED indicates that this role is authorized by the database and that no password is required to enable the role.

IDENTIFIED indicates that a user must be authorized by the specified method before the role is enabled with the SET ROLE statement:

- BY *password* creates a **local user** and indicates that the user must specify the password to Oracle when enabling the role. The password can contain only single-byte characters from your database character set regardless of whether this character set also contains multibyte characters.
- EXTERNALLY creates an **external user** and indicates that a user must be authorized by an external service (such as an operating system or third-party service) before enabling the role.

  Depending on the operating system, the user may have to specify a password to the operating system before the role is enabled.
- GLOBALLY creates a **global user** and indicates that a user must be authorized to use the role by the enterprise directory service before the role is enabled with the SET ROLE statement, or at login.

If you omit both the NOT IDENTIFIED clause and the IDENTIFIED clause, the role defaults to NOT IDENTIFIED.

Examples

The following statement creates global role VENDOR:

```
CREATE ROLE vendor IDENTIFIED GLOBALLY;
```

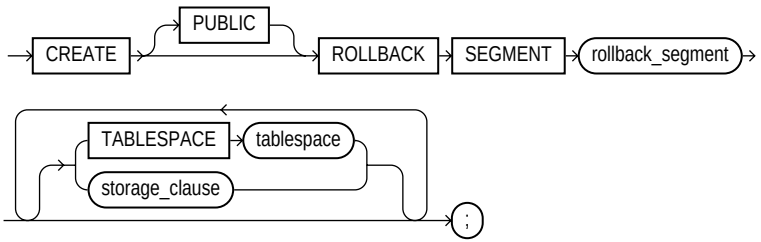
The following statement creates the role TELLER:

```
CREATE ROLE teller
   IDENTIFIED BY cashflow;
```

Users who are subsequently granted the TELLER role must specify the password CASHFLOW to enable the role with the SET ROLE statement.

CREATE ROLLBACK SEGMENT

Syntax



storage_clause: See "storage_clause" on page 7-575.

Purpose

To create a rollback segment. A **rollback segment** is an object that Oracle uses to store data necessary to reverse, or undo, changes made by transactions.

For information on altering a rollback segment, see "ALTER ROLLBACK SEGMENT" on page 7-73. For information on removing a rollback segment, see "DROP ROLLBACK SEGMENT" on page 7-470.

Prerequisites

You must have CREATE ROLLBACK SEGMENT system privilege.

Keyword and Parameters

PUBLIC	specifies that the rollback segment is public and is available to any instance. If you omit this clause, the rollback segment is private and is available only to the instance naming it in its initialization parameter ROLLBACK_SEGMENTS.
rollback_segment	is the name of the rollback segment to be created.
TABLESPACE	identifies the tablespace in which the rollback segment is created. If you omit this clause, Oracle creates the rollback segment in the SYSTEM tablespace. **Restriction:** You cannot create a rollback segment in a tablespace that is system managed (that is, during creation you specified EXTENT MANAGEMENT LOCAL AUTOALLOCATE). See "CREATE TABLESPACE" on page 7-394.

Notes:

- A tablespace can have multiple rollback segments. Generally, multiple rollback segments improve performance.
- The tablespace must be online for you to add a rollback segment to it.
- When you create a rollback segment, it is initially offline. To make it available for transactions by your Oracle instance, bring it online using the ALTER ROLLBACK SEGMENT statement. To bring it online automatically whenever you start up the database, add the segment's name to the value of the ROLLBACK_SEGMENTS initialization parameter.

For more information on creating rollback segments and making them available, see *Oracle8i Administrator's Guide*.

storage_clause specifies the characteristics for the rollback segment. See the "storage_clause" on page 7-575.

Notes:

- The OPTIMAL parameter of the *storage_clause* is of particular interest, because it applies only to rollback segments.
- You cannot specify the PCTINCREASE parameter of the *storage_clause* with CREATE ROLLBACK SEGMENT.

Examples

The following statement creates a rollback segment with default storage values in the system tablespace:

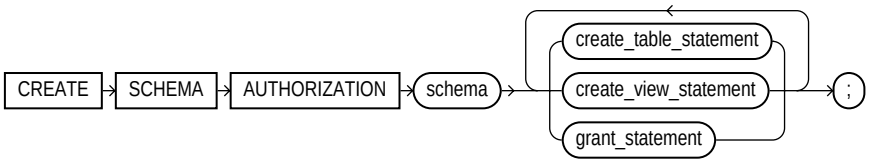
```
CREATE ROLLBACK SEGMENT rbs_2
   TABLESPACE system;
```

The above statement is equivalent to the following:

```
CREATE ROLLBACK SEGMENT rbs_2
   TABLESPACE system
   STORAGE
   ( INITIAL 10K
     NEXT 10K
     MAXEXTENTS UNLMIITED);
```

CREATE SCHEMA

Syntax



Purpose

To create multiple tables and views and perform multiple grants in a single transaction.

To execute a CREATE SCHEMA statement, Oracle executes each included statement. If all statements execute successfully, Oracle commits the transaction. If any statement results in an error, Oracle rolls back all the statements.

> **Note:** This statement does not actually create a schema. Oracle automatically creates a schema when you create a user (see "CREATE USER" on page 7-425). This statement lets you populate your schema with tables and views and grant privileges on those objects without having to issue multiple SQL statements in multiple transactions.

Prerequisites

The CREATE SCHEMA statement can include CREATE TABLE, CREATE VIEW, and GRANT statements. To issue a CREATE SCHEMA statement, you must have the privileges necessary to issue the included statements.

Keyword and Parameters

schema	is the name of the schema. The schema name must be the same as your Oracle username.
create_table_statement	is a CREATE TABLE statement to be issued as part of this CREATE SCHEMA statement. See "CREATE TABLE" on page 7-359. Do not end this statement with a semicolon (or other terminator character).

create_view_statement	is a CREATE VIEW statement to be issued as part of this CREATE SCHEMA statement. See "CREATE VIEW" on page 7-430. Do not end this statement with a semicolon (or other terminator character).
grant_statement	is a GRANT *object_privileges* statement to be issued as part of this CREATE SCHEMA statement. See "GRANT object_privileges" on page 7-505. Do not end this statement with a semicolon (or other terminator character).

The CREATE SCHEMA statement supports the syntax of these statements only as defined by standard SQL, rather than the complete syntax supported by Oracle.

The order in which you list the CREATE TABLE, CREATE VIEW, and GRANT statements is unimportant. The statements within a CREATE SCHEMA statement can reference existing objects or objects you create in other statements within the same CREATE SCHEMA statement.

Restriction: The syntax of the *parallel_clause* is allowed for a CREATE TABLE statement in CREATE SCHEMA, but parallelism is **not** used when creating the objects. For more information, see the *parallel_clause* of "CREATE TABLE" on page 7-359.

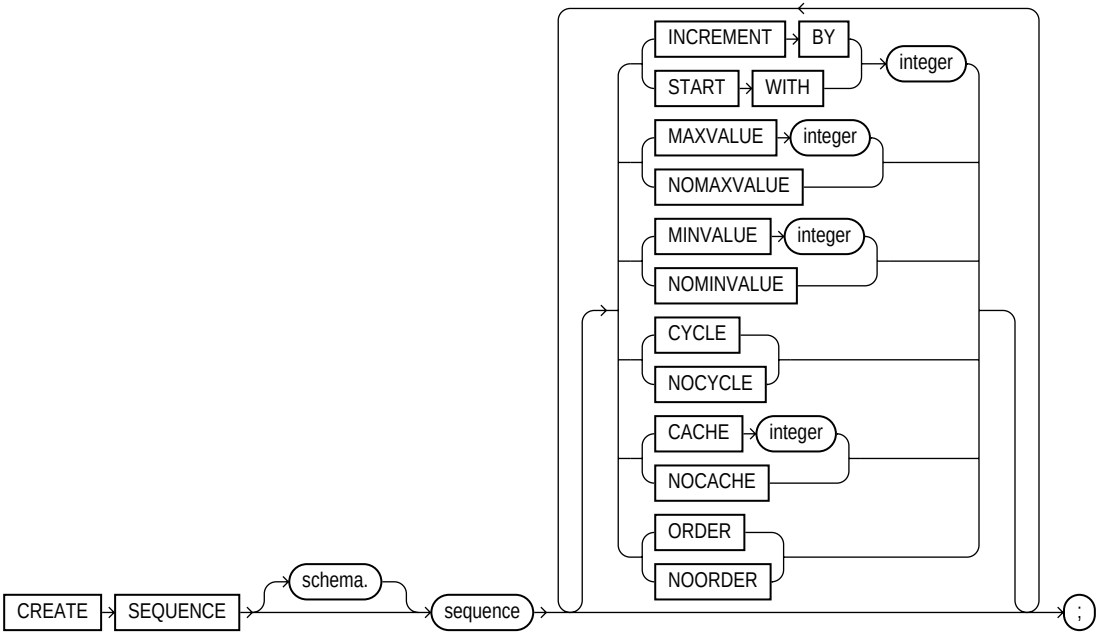
Example

The following statement creates a schema named BLAIR for the user BLAIR, creates the table SOX, creates the view RED_SOX, and grants SELECT privilege on the RED_SOX view to the user WAITES.

```
CREATE SCHEMA AUTHORIZATION blair
   CREATE TABLE sox
      (color VARCHAR2(10)  PRIMARY KEY, quantity NUMBER)
   CREATE VIEW red_sox
      AS SELECT color, quantity FROM sox WHERE color = 'RED'
   GRANT select ON red_sox TO waites;
```

CREATE SEQUENCE

Syntax



Purpose

To create a sequence. A **sequence** is a database object from which multiple users may generate unique integers. You can use sequences to automatically generate primary key values.

When a sequence number is generated, the sequence is incremented, independent of the transaction committing or rolling back. If two users concurrently increment the same sequence, the sequence numbers each user acquires may have gaps because sequence numbers are being generated by the other user. One user can never acquire the sequence number generated by another user. Once a sequence value is generated by one user, that user can continue to access that value regardless of whether the sequence is incremented by another user.

Sequence numbers are generated independently of tables, so the same sequence can be used for one or for multiple tables. It is possible that individual sequence numbers will appear to be skipped, because they were generated and used in a transaction that ultimately rolled back. Additionally, a single user may not realize that other users are drawing from the same sequence.

Once a sequence is created, you can access its values in SQL statements with the CURRVAL pseudocolumn (which returns the current value of the sequence) or the NEXTVAL pseudocolumn (which increments the sequence and returns the new value). For more information on using the above pseudocolumns, see the section "Pseudocolumns" on page 2-51.

For information on modifying or dropping a sequence, see "ALTER SEQUENCE" on page 7-76 or "DROP SEQUENCE" on page 7-471.

Prerequisites

To create a sequence in your own schema, you must have CREATE SEQUENCE privilege.

To create a sequence in another user's schema, you must have CREATE ANY SEQUENCE privilege.

Keywords and Parameters

schema is the schema to contain the sequence. If you omit *schema*, Oracle creates the sequence in your own schema.

sequence is the name of the sequence to be created.

If you specify none of the following clauses, you create an ascending sequence that starts with 1 and increases by 1 with no upper limit. Specifying only INCREMENT BY -1 creates a descending sequence that starts with -1 and decreases with no lower limit.

- **To create a sequence that increments without bound**, for ascending sequences, omit the MAXVALUE parameter or specify NOMAXVALUE. For descending sequences, omit the MINVALUE parameter or specify the NOMINVALUE.
- **To create a sequence that stops at a predefined limit**, for an ascending sequence specify a value for the MAXVALUE parameter. For a descending sequence specify a value for the MINVALUE parameter. Also specify the NOCYCLE. Any attempt to generate a sequence number once the sequence has reached its limit results in an error.
- **To create a sequence that restarts after reaching a predefined limit**, specify values for both the MAXVALUE and MINVALUE parameters. Also specify the CYCLE. If you do not specify MINVALUE, then it defaults to NOMINVALUE (that is, the value 1).

INCREMENT BY	specifies the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. This value can have 28 or fewer digits. The absolute of this value must be less than the difference of MAXVALUE and MINVALUE. If this value is negative, then the sequence descends. If the increment is positive, then the sequence ascends. If you omit this clause, the interval defaults to 1.
START WITH	specifies the first sequence number to be generated. Use this clause to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum. For ascending sequences, the default value is the sequence's minimum value. For descending sequences, the default value is the sequence's maximum value. This integer value can have 28 or fewer digits. **Note:** This value is not necessarily the value to which an ascending cycling sequence cycles after reaching its maximum or minimum value.
MAXVALUE	specifies the maximum value the sequence can generate. This integer value can have 28 or fewer digits. MAXVALUE must be equal to or greater than START WITH and must be greater than MINVALUE.
NOMAXVALUE	specifies a maximum value of 10^{27} for an ascending sequence or -1 for a descending sequence. This is the default.
MINVALUE	specifies the sequence's minimum value. This integer value can have 28 or fewer digits. MINVALUE must be less than or equal to START WITH and must be less than MAXVALUE.
NOMINVALUE	specifies a minimum value of 1 for an ascending sequence or $-(10^{26})$ for a descending sequence. This is the default.
CYCLE	specifies that the sequence continues to generate values after reaching either its maximum or minimum value. After an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum, it generates its maximum.
NOCYCLE	specifies that the sequence cannot generate more values after reaching its maximum or minimum value. This is the default.
CACHE	specifies how many values of the sequence Oracle preallocates and keeps in memory for faster access. This integer value can have 28 or fewer digits. The minimum value for this parameter is 2. For sequences that cycle, this value must be less than the number of values in the cycle. You cannot cache more values than will fit in a given cycle of sequence numbers. Therefore, the maximum value allowed for CACHE must be less than the value determined by the following formula: `(CEIL (MAXVALUE - MINVALUE)) / ABS (INCREMENT)` If a system failure occurs, all cached sequence values that have not been used in committed DML statements are lost. The potential number of lost values is equal to the value of the CACHE parameter.
NOCACHE	specifies that values of the sequence are not preallocated.

If you omit both CACHE and NOCACHE, Oracle caches 20 sequence numbers by default.

ORDER	guarantees that sequence numbers are generated in order of request. You may want to use this clause if you are using the sequence numbers as timestamps. Guaranteeing order is usually not important for sequences used to generate primary keys. ORDER is necessary only to guarantee ordered generation if you are using Oracle with the Parallel Server option in parallel mode. If you are using exclusive mode, sequence numbers are always generated in order.
NOORDER	does not guarantee sequence numbers are generated in order of request. This is the default.

Example

The following statement creates the sequence ESEQ:

```
CREATE SEQUENCE eseq
  INCREMENT BY 10;
```

The first reference to ESEQ.NEXTVAL returns 1. The second returns 11. Each subsequent reference will return a value 10 greater than the one previous.

CREATE SNAPSHOT

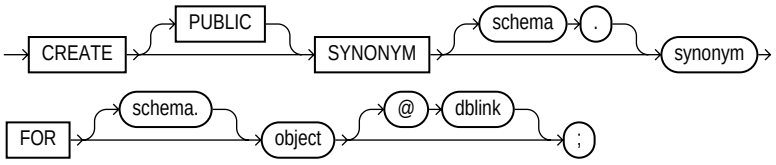
In Oracle8*i*, "snapshots" are synonymous with "materialized views." Please see "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 7-300.

CREATE SNAPSHOT LOG

In Oracle8*i*, "snapshots" are synonymous with "materialized views." Please see "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 7-300.

CREATE SYNONYM

Syntax



Purpose

To create a synonym. A *synonym* is an alternative name for a table, view, sequence, procedure, stored function, package, materialized view, Java class schema object, or another synonym. For general information on synonyms, see *Oracle8i Concepts*.

Synonyms provide both data independence and location transparency. Synonyms permit applications to function without modification regardless of which user owns the table or view and regardless of which database holds the table or view.

Table 7–4 lists the SQL statements in which you can refer to synonyms.

Table 7–4 Using Synonyms

DML Statements	DDL Statements
SELECT	AUDIT
INSERT	NOAUDIT
UPDATE	GRANT
DELETE	REVOKE
EXPLAIN PLAN	COMMENT
LOCK TABLE	

Prerequisites

To create a private synonym in your own schema, you must have CREATE SYNONYM system privilege.

To create a private synonym in another user's schema, you must have CREATE ANY SYNONYM system privilege.

To create a PUBLIC synonym, you must have CREATE PUBLIC SYNONYM system privilege.

Keywords and Parameters

PUBLIC — creates a public synonym. Public synonyms are accessible to all users.

Oracle uses a public synonym only when resolving references to an object if the object is not prefaced by a schema and the object is not followed by a database link.

If you omit this clause, the synonym is private and is accessible only within its schema. A private synonym name must be unique in its schema.

schema — is the schema to contain the synonym. If you omit *schema*, Oracle creates the synonym in your own schema. You cannot specify schema if you have specified PUBLIC.

synonym — is the name of the synonym to be created.

FOR — identifies the object for which the synonym is created. If you do not qualify object with *schema*, Oracle assumes that the schema object is in your own schema. The schema object can be of the following types:

- table or object table
- view or object view
- sequence
- stored procedure, function, or package
- materialized view
- Java class schema object
- synonym

The schema object need not currently exist and you need not have privileges to access the object.

Restrictions:

- The schema object cannot be contained in a package.
- You cannot create a synonym for an object type.

dblink — You can use a complete or partial *dblink* to create a synonym for a schema object on a remote database where the object is located. For more information on referring to database links, see "Referring to Objects in Remote Databases" on page 2-74. If you specify *dblink* and omit *schema*, the synonym refers to an object in the schema specified by the database link. Oracle Corporation recommends that you specify the schema containing the object in the remote database.

For more information on database links, see "CREATE DATABASE LINK" on page 7-255.

If you omit *dblink*, Oracle assumes the object is located on the local database.

Restriction: You cannot specify *dblink* for a Java class synonym.

Examples

Oracle attempts to resolve references to objects at the schema level before resolving them at the PUBLIC synonym level. For example, assume the schemas SCOTT and BLAKE each contain tables named DEPT and the user SYSTEM creates a PUBLIC synonym named DEPT for BLAKE.DEPT. If the user SCOTT then issues the following statement, Oracle returns rows from SCOTT.DEPT:

```
SELECT * FROM dept;
```

To retrieve rows from BLAKE.DEPT, the user SCOTT must preface DEPT with the schema name:

```
SELECT * FROM blake.dept;
```

If the user ADAM's schema does not contain an object named DEPT, then ADAM can access the DEPT table in BLAKE's schema by using the public synonym DEPT:

```
SELECT * FROM dept;
```

To define the synonym MARKET for the table MARKET_RESEARCH in the schema SCOTT, issue the following statement:

```
CREATE SYNONYM market
   FOR scott.market_research;
```

To create a PUBLIC synonym for the EMP table in the schema SCOTT on the remote SALES database, you could issue the following statement:

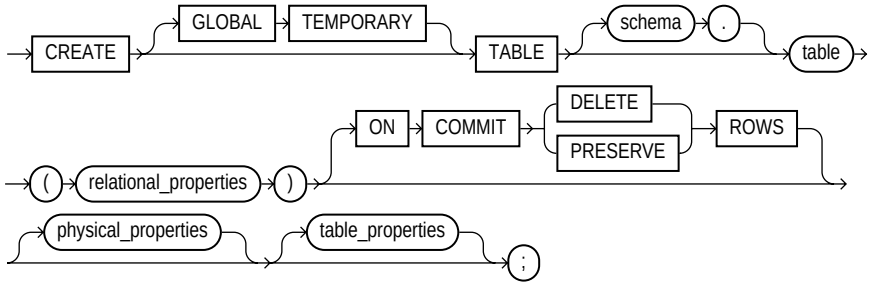
```
CREATE PUBLIC SYNONYM emp
   FOR scott.emp@sales;
```

A synonym may have the same name as the base table, provided the base table is contained in another schema.

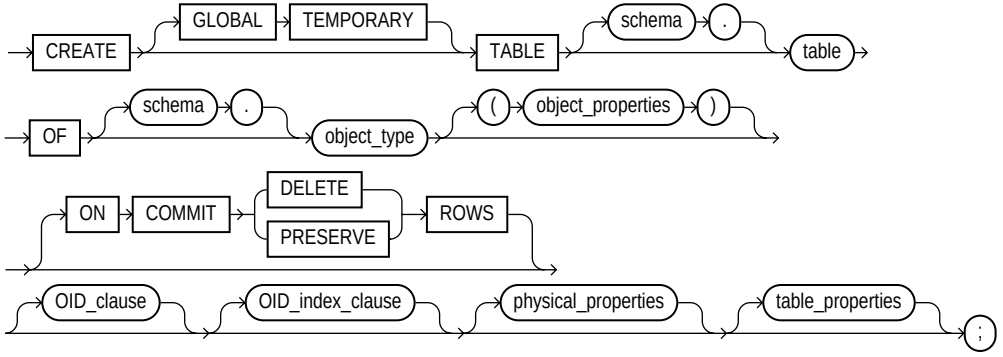
CREATE TABLE

Syntax

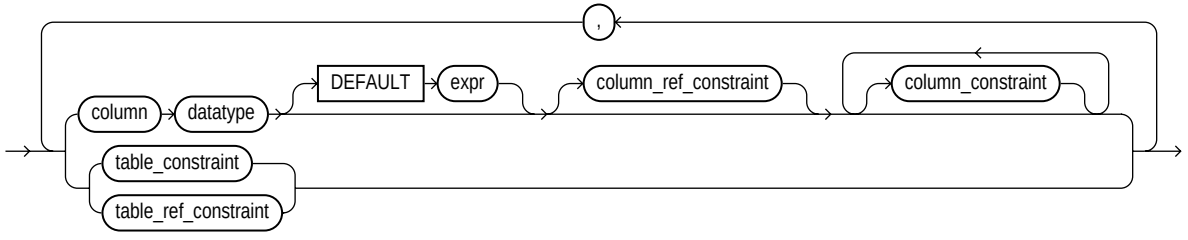
relational_table:



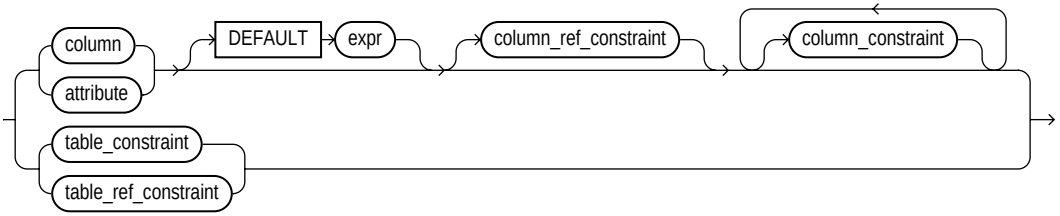
object_table:



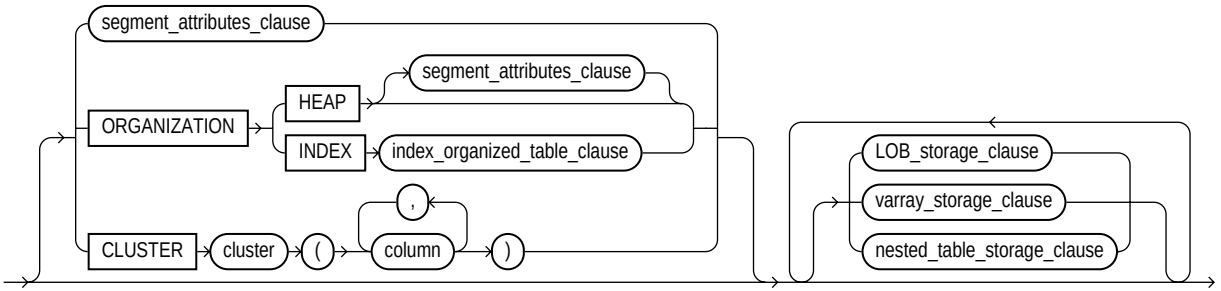
relational_properties::=



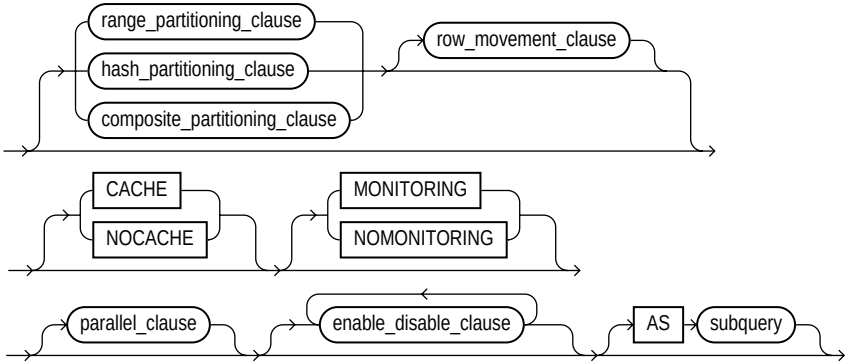
object properties::=



physical_properties::=



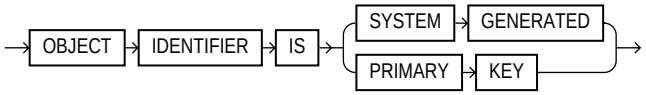
table_properties::=



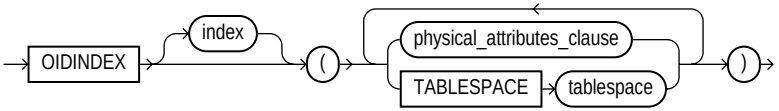
subquery::= See "SELECT and Subqueries" on page 7-541.

table_constraint, column_constraint, table_ref_constraint, column_ref_constraint, constraint_state: See the "constraint_clause" on page 7-217

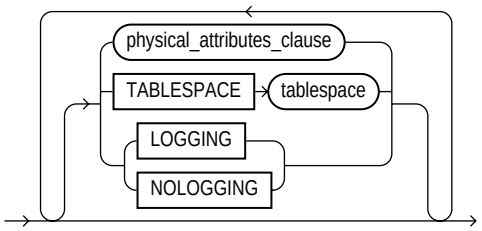
OID_clause::=



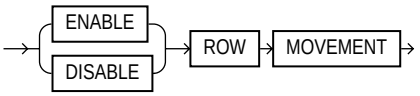
OID_index_clause::=



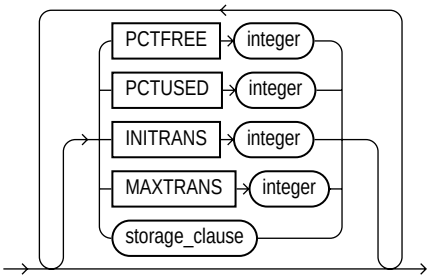
segment_attributes_clause::=



row_movement_clause::=

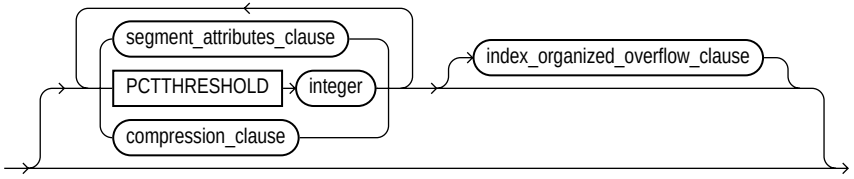


physical_attributes_clause::=

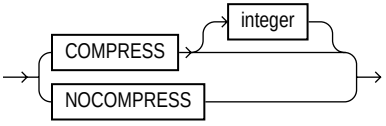


storage_clause: See the "storage_clause" on page 7-575.

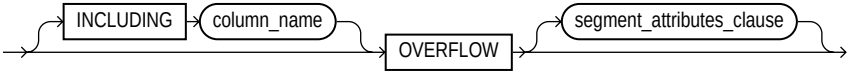
index_organized_table_clause::=



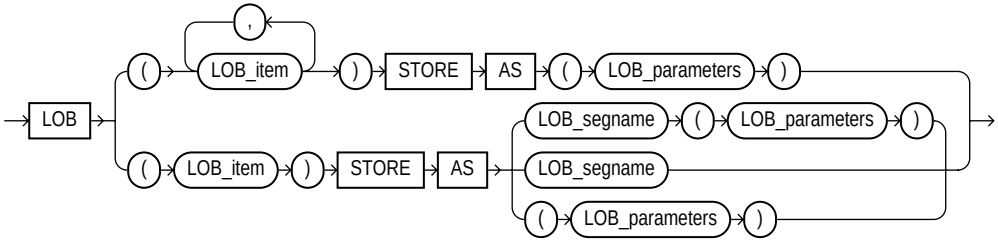
compression_clause::=



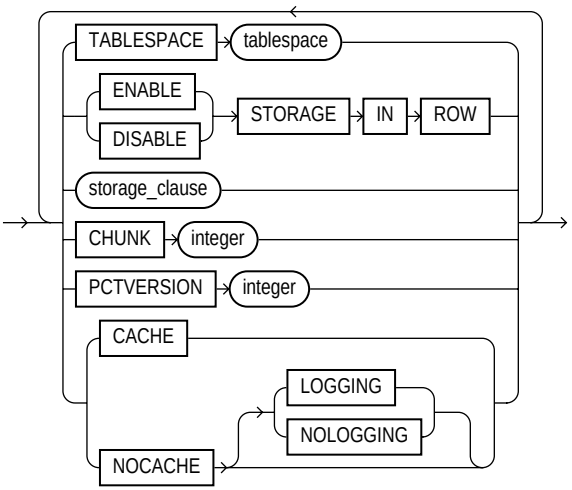
index_organized_overflow_clause::=



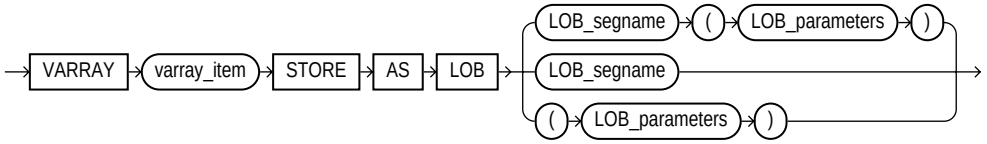
LOB_storage_clause::=



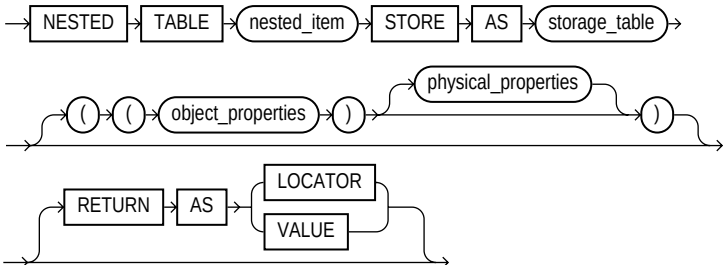
LOB_parameters::=



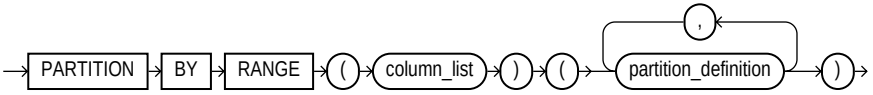
varray_storage_clause::=



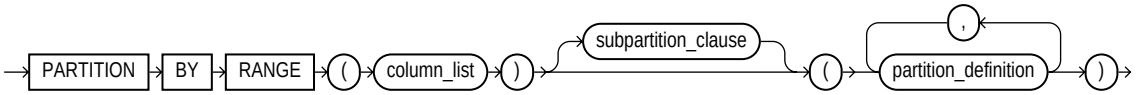
nested_table_storage_clause::=



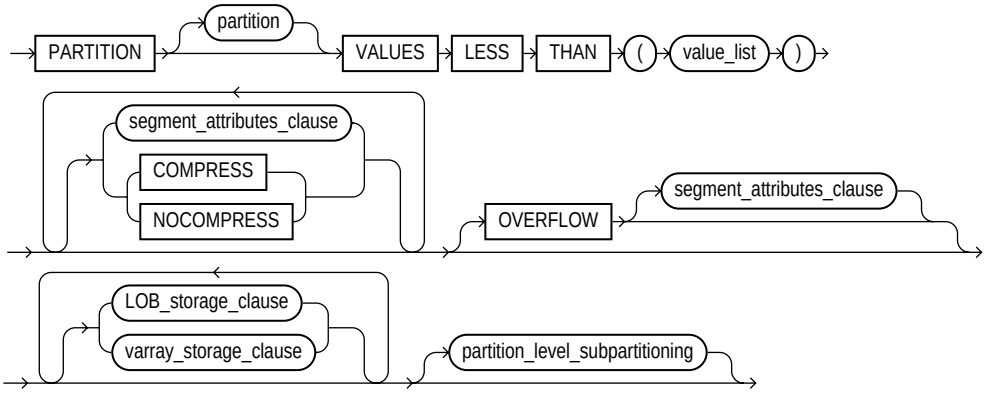
range_partitioning_clause::=



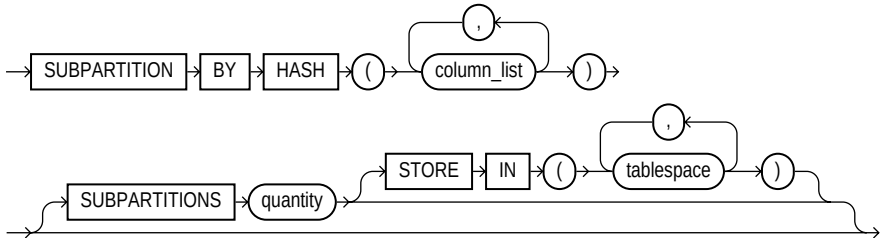
composite_partitioning_clause::=



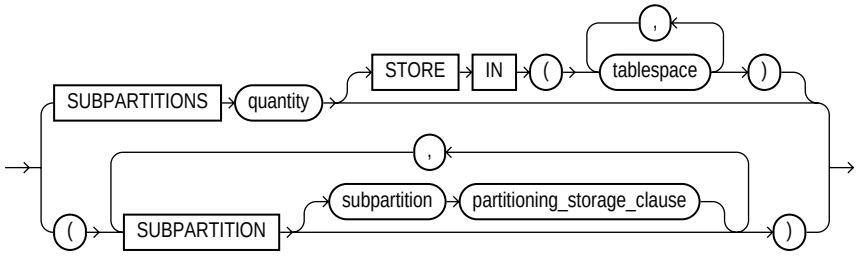
partition_definition::=



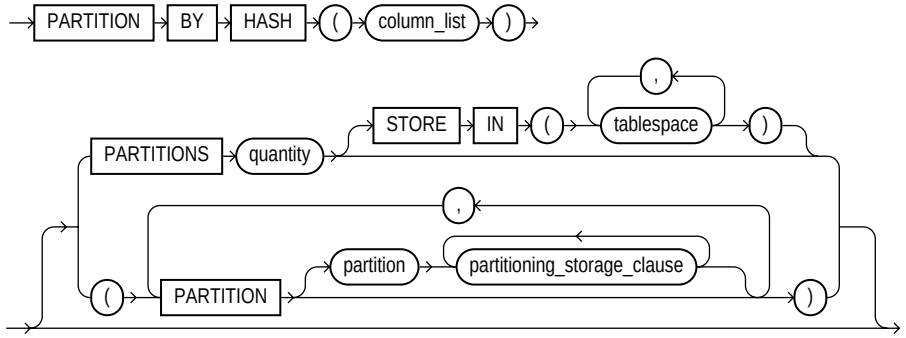
subpartition_clause::=



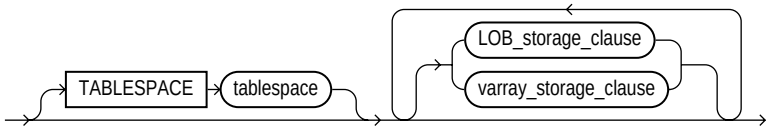
partition_level_subpartitioning::=



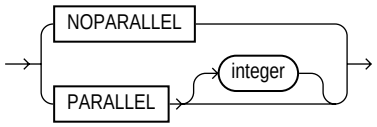
hash_partitioning_clause::=



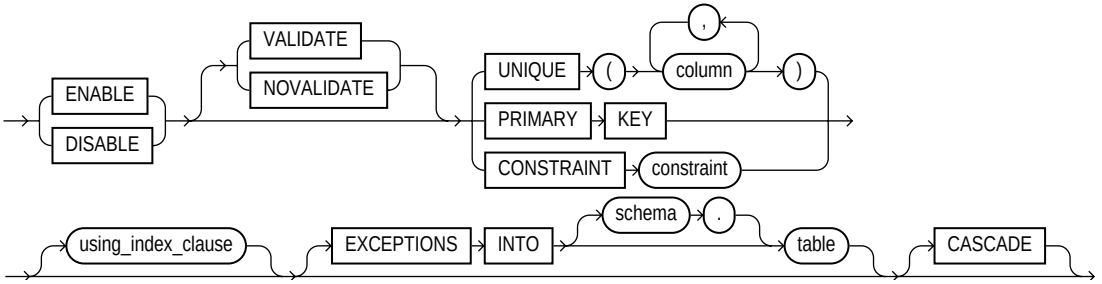
partitioning_storage_clause::=



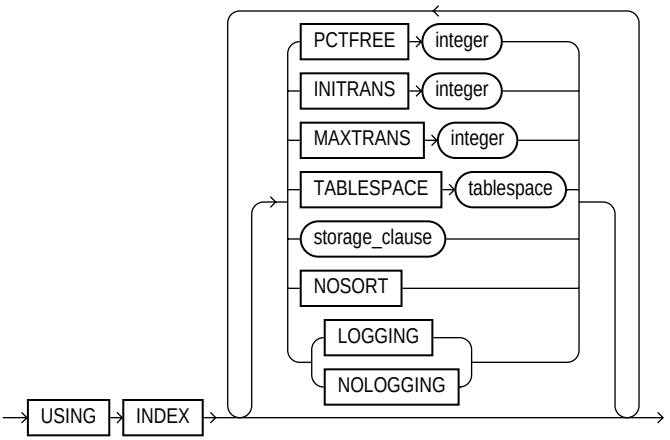
parallel_clause::=



enable_disable_clause::=



using_index_clause::=



Purpose

To create a **relational table**, the basic structure to hold user data.

To create an **object table** or a table that uses an object type for a column definition. An object table is a table explicitly defined to hold object instances of a particular type.

You can also create an object type and then use it in a column when creating a relational table. For more information about creating objects, see *Oracle8i Application Developer's Guide - Fundamentals* and "CREATE TYPE" on page 7-411.

Tables are created with no data unless a query is specified. You can add rows to a table with the INSERT statement. After creating a table, you can define additional columns, partitions, and integrity constraints with the ADD clause of the ALTER TABLE statement. You can change the definition of an existing column or partition with the MODIFY clause of the ALTER TABLE statement.

Prerequisites

To create a relational table in your own schema, you must have CREATE TABLE system privilege. To create a table in another user's schema, you must have CREATE ANY TABLE system privilege. Also, the owner of the schema to contain the table must have either space quota on the tablespace to contain the table or UNLIMITED TABLESPACE system privilege.

In addition to the table privileges above, to create a table that uses types, the owner of the table must have the EXECUTE object privilege in order to access all types referenced by the table, or you must have the EXECUTE ANY TYPE system privilege. These privileges must be granted explicitly and not acquired through a role.

Additionally, if the table owner intends to grant access to the table to other users, the owner must have been granted the EXECUTE privileges to the referenced types with the GRANT OPTION, or have the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. Without these privileges, the table owner has insufficient privileges to grant access on the table to other users.

To enable a UNIQUE or PRIMARY KEY constraint, you must have the privileges necessary to create an index on the table. You need these privileges because Oracle creates an index on the columns of the unique or primary key in the schema containing the table. See "CREATE INDEX" on page 7-273.

For more information about the privileges required to create tables using types, see *Oracle8i Application Developer's Guide - Fundamentals*.

Keywords and Parameters

GLOBAL TEMPORARY

specifies that the table is temporary and that its definition is visible to all sessions. The data in a temporary table is visible only to the session that inserts the data into the table.

A temporary table has a definition that persists the same as the definitions of regular tables, but it contains either **session-specific** or **transaction-specific** data. You specify whether the data is session- or transaction-specific with the ON COMMIT keywords (below).

For more information on temporary tables, please refer to *Oracle8i Concepts*.

Restrictions:

- Temporary tables cannot be partitioned, index-organized, or clustered.
- You cannot specify any referential integrity (foreign key) constraints on temporary tables.
- Temporary tables cannot contain columns of nested table or varray type.
- You cannot specify the following clauses of the *LOB_storage_clause*: TABLESPACE, *storage_clause*, LOGGING or NOLOGGING, MONITORING or NOMONITORING, or *LOB_index_clause*.
- Parallel DML and parallel queries are not supported for temporary tables. (Parallel hints are ignored. Specification of the *parallel_clause* returns an error.)
- You cannot specify the *segment_attributes_clause*, *nested_table_storage_clause*, or *parallel_clause*.
- Distributed transactions are not supported for temporary tables.

schema

is the schema to contain the table. If you omit *schema*, Oracle creates the table in your own schema.

table

is the name of the table (or object table) to be created. A partitioned *table* cannot be a clustered table or an object table.

OF *object_type*

explicitly creates an object table of type *object_type*. The columns of an object table correspond to the top-level attributes of type *object_type*. Each row will contain an object instance, and each instance will be assigned a unique, system-generated object identifier (OID) when a row is inserted. If you omit schema, Oracle creates the object table in your own schema. For more information about creating objects, see "CREATE TYPE" on page 7-411.

Objects residing in an object table are referenceable. For more information about using REFs, see "User-Defined Type Categories" on page 2-25, "User-Defined Functions" on page 4-56, "Expressions" on page 5-1, "CREATE TYPE" on page 7-411, and *Oracle8i Administrator's Guide*.

column specifies the name of a column of the table.

If you also specify AS *subquery*, you can omit *column* and *datatype* unless you are creating an index-organized table (IOT). If you specify AS *subquery* when creating an IOT, you must specify *column*, and you must omit *datatype*.

The absolute maximum number of columns in a table is 1000. However, when you create an object table (or a relational table with columns of object, nested table, varray, or REF type), Oracle maps the columns of the user-defined types to relational columns, creating in effect "hidden columns" that count toward the 1000-column limit. For details on how Oracle calculates the total number of columns in such a table, please refer to *Oracle8i Administrator's Guide*.

attribute specifies the qualified column name of an item in an object.

datatype is the datatype of a column. Oracle-supplied datatypes are defined in "Datatypes" on page 2-5.

Restrictions:

- You cannot specify a LOB column or a column of type VARRAY for a partitioned index-organized table. The datatypes for nonpartitioned index-organized tables are not restricted.
- You can specify a column of type ROWID, but Oracle does not guarantee that the values in such columns are valid rowids.

You can omit *datatype*:

- If you also specify AS *subquery*. (If you are creating an index-organized table and you specify AS *subquery*, you **must** omit the datatype.)
- If the statement also designates the column as part of a foreign key in a referential integrity constraint. (Oracle automatically assigns to the column the datatype of the corresponding column of the referenced key of the referential integrity constraint.)

DEFAULT specifies a value to be assigned to the column if a subsequent INSERT statement omits a value for the column. The datatype of the expression must match the datatype of the column. The column must also be long enough to hold this expression. For the syntax of *expr*, see "Expressions" on page 5-1. A DEFAULT expression cannot contain references to other columns, the pseudocolumns CURRVAL, NEXTVAL, LEVEL, and ROWNUM, or date constants that are not fully specified.

table_ref_constraint

and

column_ref_constraint

These clauses let you further describe a column of type REF. The only difference between these clauses is that you specify *table_ref* from the table level, so you must identify the REF column or attribute you are defining. You specify *column_ref* after you have already identified the REF column or attribute.

For syntax and description of these constraints, see the "constraint_clause" on page 7-217.

column_constraint defines an integrity constraint as part of the column definition. See the syntax description of *column_constraint* in the "constraint_clause" on page 7-217.

You can create UNIQUE, PRIMARY KEY, and REFERENCES constraints on scalar attributes of object type columns. You can also create NOT NULL constraints on object type columns, and CHECK constraints that reference object type columns or any attribute of an object type column.

table_constraint defines an integrity constraint as part of the table definition. See the syntax description of *table_constraint* in the "constraint_clause" on page 7-217.

Note: You must specify a PRIMARY KEY constraint for an index-organized table.

segment_attributes_clause:

physical_ attributes_clause specifies the value of the PCTFREE, PCTUSED, INITRANS, and MAXTRANS parameters and the storage characteristics of the table.

- For a nonpartitioned table, each parameter and storage characteristic you specify determines the actual physical attribute of the segment associated with the table.
- For partitioned tables, the value you specify for the parameter or storage characteristic is the default physical attribute of the segments associated with all partitions specified in this CREATE statement (and in subsequent ALTER TABLE ... ADD PARTITION statements), unless you explicitly override that value in the PARTITION clause of the statement that creates the partition.

PCTFREE specifies the percentage of space in each data block of the table, object table OID index, or partition reserved for future updates to the table's rows. The value of PCTFREE must be a value from 0 to 99. A value of 0 allows the entire block to be filled by inserts of new rows. The default value is 10. This value reserves 10% of each block for updates to existing rows and allows inserts of new rows to fill a maximum of 90% of each block.

PCTFREE has the same function in the PARTITION description and in the statements that create and alter clusters, indexes, snapshots, and snapshot logs. The combination of PCTFREE and PCTUSED determines whether new rows will be inserted into existing data blocks or into new blocks.

PCTUSED specifies the minimum percentage of used space that Oracle maintains for each data block of the table, object table OID index, or index-organized table overflow data segment. A block becomes a candidate for row insertion when its used space falls below PCTUSED. PCTUSED is specified as a positive integer from 1 to 99 and defaults to 40.

PCTUSED has the same function in the PARTITION description and in the statements that create and alter clusters, snapshots, and snapshot logs.

PCTUSED is not a valid table storage characteristic for an index-organized table (ORGANIZATION INDEX).

The sum of PCTFREE and PCTUSED must be less than 100. You can use PCTFREE and PCTUSED together to utilize space within a table more efficiently. For information on the performance effects of different values PCTUSED and PCTFREE, see *Oracle8i Tuning*.

INITRANS — specifies the initial number of transaction entries allocated within each data block allocated to the table, object table OID index, partition, LOB index segment, or overflow data segment. This value can range from 1 to 255 and defaults to 1. In general, you should not change the INITRANS value from its default.

Each transaction that updates a block requires a transaction entry in the block. The size of a transaction entry depends on your operating system.

This parameter ensures that a minimum number of concurrent transactions can update the block and helps avoid the overhead of dynamically allocating a transaction entry.

The INITRANS parameter serves the same purpose in the PARTITION description, clusters, indexes, snapshots, and snapshot logs as in tables. The minimum and default INITRANS value for a cluster or index is 2, rather than 1.

MAXTRANS — specifies the maximum number of concurrent transactions that can update a data block allocated to the table, object table OID index, partition, LOB index segment, or index-organized overflow data segment. This limit does not apply to queries. This value can range from 1 to 255 and the default is a function of the data block size. You should not change the MAXTRANS value from its default.

If the number of concurrent transactions updating a block exceeds the INITRANS value, Oracle dynamically allocates transaction entries in the block until either the MAXTRANS value is exceeded or the block has no more free space.

The MAXTRANS parameter serves the same purpose in the PARTITION description, clusters, snapshots, and snapshot logs as in tables.

storage_clause — specifies the storage characteristics for the table, object table OID index, partition, LOB storage, LOB index segment, or index-organized table overflow data segment. This clause has performance ramifications for large tables. Storage should be allocated to minimize dynamic allocation of additional space. See the "storage_clause" on page 7-575.

TABLESPACE — specifies the tablespace in which Oracle creates the table, object table OID index, partition, LOB storage, LOB index segment, or index-organized table overflow data segment. If you omit TABLESPACE, then Oracle creates that item in the default tablespace of the owner of the schema containing the table.

For heap-organized tables with one or more LOB columns, if you omit the TABLESPACE clause for LOB storage, Oracle creates the LOB data and index segments in the tablespace where the table is created.

However, for an index-organized table with one or more LOB columns, if you omit TABLESPACE, the LOB data and index segments are created in the tablespace in which the primary key index segment of the index-organized table is created.

For nonpartitioned tables, the value specified for TABLESPACE is the actual physical attribute of the segment associated with the table. For partitioned tables, the value specified for TABLESPACE is the default physical attribute of the segments associated with all partitions specified in the CREATE statement (and on subsequent ALTER TABLE ... ADD PARTITION statements), unless you specify TABLESPACE in the PARTITION description.

For more information on tablespaces, see "CREATE TABLESPACE" on page 7-394.

LOGGING | NOLOGGING

specifies whether the creation of the table (and any indexes required because of constraints), partition, or LOB storage characteristics will be logged in the redo log file.The logging attribute of the table is independent of that of its indexes.

This attribute also specifies that subsequent Direct Loader (SQL*Loader) and direct-load INSERT operations against the table, partition, or LOB storage are logged (LOGGING) or not logged (NOLOGGING).

If you omit LOGGING | NOLOGGING, the logging attribute of the table or table partition defaults to the logging attribute of the tablespace in which it resides. For LOBs, if you omit LOGGING | NOLOGGING,

- If you specify CACHE, then LOGGING is used (because you cannot have CACHE NOLOGGING).
- Otherwise, the logging attribute defaults to the logging attribute of the tablespace in which it resides.

For nonpartitioned tables, the value specified for LOGGING is the actual physical attribute of the segment associated with the table. For partitioned tables, the logging attribute value specified is the default physical attribute of the segments associated with all partitions specified in the CREATE statement (and in subsequent ALTER TABLE ... ADD PARTITION statements), unless you specify LOGGING | NOLOGGING in the PARTITION description.

In NOLOGGING mode, data is modified with minimal logging (to mark new extents INVALID and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not fully logged. Therefore, if you cannot afford to lose this table, you should take a backup after the NOLOGGING operation.

The size of a redo log generated for an operation in NOLOGGING mode is significantly smaller than the log generated with the LOGGING attribute set.

If the database is run in ARCHIVELOG mode, media recovery from a backup taken before the LOGGING operation restores the table. However, media recovery from a backup taken before the NOLOGGING operation does not restore the table.

For more information about logging and parallel DML, see *Oracle8i Concepts* and *Oracle8i Administrator's Guide*.

RECOVERABLE | UNRECOVERABLE

These keywords are deprecated and have been replaced with LOGGING and NOLOGGING, respectively. Although RECOVERABLE and UNRECOVERABLE are supported for backward compatibility, Oracle Corporation strongly recommends that you use the LOGGING and NOLOGGING keywords.

Restrictions:

- You cannot specify RECOVERABLE for partitioned tables or LOB storage characteristics.
- You cannot specify UNRECOVERABLE for a partitioned or index-organized tables.
- You can specify UNRECOVERABLE only with AS *subquery*.

ORGANIZATION HEAP	specifies that the data rows of *table* are stored in no particular order. This is the default.
ORGANIZATION INDEX	specifies that *table* is created as an index-organized table. In an index-organized table, the data rows are held in an index defined on the primary key for the table.
index_organized_table_clause	specifies that Oracle should maintain the table rows (both primary key column values and non-key column values) in a B*-tree index built on the primary key. Index-organized tables are therefore best suited for primary key-based access and manipulation. An index-organized table is an alternative to ■ A nonclustered table indexed on the primary key by using the CREATE INDEX statement ■ A clustered table stored in an indexed cluster that has been created using the CREATE CLUSTER statement that maps the primary key for the table to the cluster key **Restrictions**: ■ You cannot specify a column of type ROWID for an index-organized table. ■ A partitioned index-organized table cannot contain columns of LOB or varray type. (This restriction does not apply to nonpartitioned index-organized tables.) **Note**: You must specify a primary key for an index-organized table, because the primary key uniquely identifies a row. Use the primary key instead of the rowid for directly accessing index-organized rows.
PCTTHRESHOLD *integer*	specifies the percentage of space reserved in the index block for an index-organized table row. Any portion of the row that exceeds the specified threshold is stored in the overflow segment. PCTTHRESHOLD must be a value from 1 to 50. **Restriction**: ■ PCTTHRESHOLD must be large enough to hold the primary key. ■ You cannot specify PCTTHRESHOLD for individual partitions of an index-organized table.
OVERFLOW	specifies that index-organized table data rows exceeding the specified threshold are placed in the data segment listed in this clause.

- When you create an index-organized table, Oracle evaluates the maximum size of each column to estimate the largest possible row. If an overflow segment is needed but you have not specified OVERFLOW, Oracle raises an error and does not execute the CREATE TABLE statement. This checking function guarantees that subsequent DML operations on the index-organized table will not fail because an overflow segment is lacking.
- All physical attributes and storage characteristics you specify in this clause after the OVERFLOW keyword apply only to the overflow segment of the table. Physical attributes and storage characteristics for the index-organized table itself, default values for all its partitions, and values for individual partitions must be specified before this keyword.
- If the index-organized table contains one or more LOB columns, the LOBs will be stored out-of-line unless you specify OVERFLOW, even if they would otherwise be small enough be to stored inline.

INCLUDING *column_name* specifies a column at which to divide an index-organized table row into index and overflow portions. All non-primary-key columns that follow *column_name* are stored in the overflow data segment. A *column_name* is either the name of the last primary-key column or any subsequent nonprimary-key column.

Restriction: You cannot specify this clause for individual partitions of an index-organized table.

compression_clause enables or disables key compression.

COMPRESS enables key compression, which eliminates repeated occurrence of primary key column values in index-organized tables. Use *integer* to specify the prefix length (number of prefix columns to compress).

The valid range of prefix length values is from 1 to the number of primary key columns minus 1. The default prefix length is the number of primary key columns minus 1.

Restriction: At the partition level, you can specify COMPRESS, but you cannot specify the prefix length with *integer*.

NOCOMPRESS disables key compression in index-organized tables. This is the default.

LOB_storage_clause specifies the storage attributes of LOB data segments. See also "LOB Column Example" on page 7-389.

- For a nonpartitioned table (that is, when specified in the *physical_properties* clause without any of the partitioning clauses), this clause specifies the table's storage attributes of LOB data segments.

- For a partitioned table specified at the table level (that is, when specified in the *physical_properties* clause along with one of the partitioning clauses), this clause specifies the default storage attributes for LOB data segments associated with each partition or subpartition. These storage attributes apply to all partitions or subpartitions unless overridden by a *LOB_storage_clause* at the partition or subpartition level.
- For an individual partition of a partitioned table (that is, when specified as part of a *partition_definition*), this clause specifies the storage attributes of the data segments of that partition or the default storage attributes of any subpartitions of this partition. A partition-level *LOB_storage_clause* overrides a table-level *LOB_storage_clause*.
- For an individual subpartition of a partitioned table (that is, when specified as part of a *subpartition_clause*), this clause specifies the storage attributes of the data segments of this subpartition. A subpartition-level *LOB_storage_clause* overrides both partition-level and table-level *LOB_storage_clauses*.

Restrictions:

- The only parameter of the *LOB_storage_clause* you can specify for a hash partition or hash subpartition is TABLESPACE.
- You cannot specify the *LOB_index_clause* if *table* is partitioned.

For detailed information about LOBs, see *Oracle8i Application Developer's Guide - Fundamentals*.

	lob_item	is the LOB column name or LOB object attribute for which you are explicitly defining tablespace and storage characteristics that are different from those of the table. Oracle automatically creates a system-managed index for each *lob_item* you create. **Restriction:** If table is partitioned, you cannot specify LOB storage for a LOB object attribute.
STORE AS	*lob_segname*	specifies the name of the LOB data segment. You cannot use *lob_segname* if you specify more than one *lob_item*.
lob_parameters	ENABLE STORAGE IN ROW	specifies that the LOB value is stored in the row (inline) if its length is less than approximately 4000 bytes minus system control information. This is the default. **Restriction:** For an index-organized table, you cannot specify this parameter unless you have specified an OVERFLOW segment in the *index_organized_table_clause*.
	DISABLE STORAGE IN ROW	specifies that the LOB value is stored outside of the row regardless of the length of the LOB value.

The LOB locator is always stored in the row regardless of where the LOB value is stored. You cannot change the value of STORAGE IN ROW once it is set.

	CHUNK *integer*	specifies the number of bytes to be allocated for LOB manipulation. If *integer* is not a multiple of the database block size, Oracle rounds up (in bytes) to the next multiple. For example, if the database block size is 2048 and *integer* is 2050, Oracle allocates 4096 bytes (2 blocks). The maximum value is 32768 (32K), which is the largest Oracle block size allowed. The default CHUNK size is one Oracle database block. You cannot change the value of CHUNK once it is set. **Note:** The value of CHUNK must be less than or equal to the value of NEXT (either the default value or that specified in the *storage_clause*). If CHUNK exceeds the value of NEXT, Oracle returns an error.
	PCTVERSION *integer*	is the maximum percentage of overall LOB storage space used for creating new versions of the LOB. The default value is 10, meaning that older versions of the LOB data are not overwritten until 10% of the overall LOB storage space is used.
	LOB_index_clause	This clause is deprecated as of Oracle 8*i*. Oracle generates an index for each LOB column. The LOB indexes are system named and system managed. Although it is still possible for you to specify this clause, Oracle Corporation strongly recommends that you no longer do so. For information on how Oracle manages LOB indexes in tables migrated from earlier versions, see *Oracle8i Migration*.

varray_storage_clause lets you specify separate storage characteristics for the LOB in which a varray will be stored. In addition, if you specify this clause, Oracle will always store the varray in a LOB, even if it is small enough to be stored inline.

- For a nonpartitioned table (that is, when specified in the *physical_properties* clause without any of the partitioning clauses), this clause specifies the storage attributes of the varray's LOB data segments.
- For a partitioned table specified at the table level (that is, when specified in the *physical_properties* clause along with one of the partitioning clauses), this clause specifies the default storage attributes for the varray's LOB data segments associated with each partition (or its subpartitions, if any).
- For an individual partition of a partitioned table (that is, when specified as part of a *partition_definition*), this clause specifies the storage attributes of the varray's LOB data segments of that partition or the default storage attributes of the varray's LOB data segments of any subpartitions of this partition. A partition-level *varray_storage_clause* overrides a table-level *varray_storage_clause*.
- For an individual subpartition of a partitioned table (that is, when specified as part of a *subpartition_clause*), this clause specifies the storage attributes of the varray's data segments of this subpartition. A subpartition-level *varray_storage_clause* overrides both partition-level and table-level *varray_storage_clauses*.

Restriction: You cannot specify the TABLESPACE parameter of *lob_parameters* as part of this clause. The LOB tablespace for a varray defaults to the containing table's tablespace.

nested_table_storage_clause enables you to specify separate storage characteristics for a nested table, which in turn enables you to define the nested table as an index-organized table. The storage table is created in the same tablespace as its parent table (using the default storage characteristics) and stores the nested table values of the column for which it was created.

You must include this clause when creating a table with columns or column attributes whose type is a nested table. (Clauses within this clause that function the same way they function for parent object tables are not repeated here.)

Restrictions:

- You cannot specify this clause for a temporary table.
- You cannot specify the *parallel_clause* or the the *OID_clause*.
- You cannot specify TABLESPACE (as part of the *segment_attributes_clause*) for a nested table. The tablespace is always that of the parent table.
- At create time, you cannot specify (as part of *object_properties*) a *table_ref_constraint*, *column_ref_constraint*, or referential constraint for the attributes of a nested table. However, you can modify a nested table to add such constraints using ALTER TABLE.
- You cannot query or perform DML statements on the storage table directly, but you can modify the nested table column storage characteristics by using the name of storage table in an ALTER TABLE statement. For information about modifying nested table column storage characteristics, see "ALTER TABLE" on page 7-113.

nested_item is the name of a column (or a top-level attribute of the table's object type) whose type is a nested table.

storage_table is the name of the table where the rows of *nested_item* reside. The storage table is created in the same schema and the same tablespace as the parent table.

You cannot query or perform DML statements on *storage_table* directly, but you can modify its storage characteristics by specifying its name in an ALTER TABLE statement. For information about modifying nested table column storage characteristics, see "ALTER TABLE" on page 7-113.

RETURN AS specifies what Oracle returns as the result of a query.

- VALUE returns a copy of the nested table itself.
- LOCATOR returns a collection locator to the copy of the nested table.

Note: The locator is scoped to the session and cannot be used across sessions. Unlike a LOB locator, the collection locator cannot be used to modify the collection instance.

If you do not specify the *segment_attributes_clause* or the *LOB_storage_clause*, the nested table is heap organized and is created with default storage characteristics.

CLUSTER — specifies that the table is to be part of *cluster*. The columns listed in this clause are the table columns that correspond to the cluster's columns. Generally, the cluster columns of a table are the column or columns that make up its primary key or a portion of its primary key. For more information, see "CREATE CLUSTER" on page 7-236.

Specify one column from the table for each column in the cluster key. The columns are matched by position, not by name.

A clustered table uses the cluster's space allocation. Therefore, do not use the PCTFREE, PCTUSED, INITRANS, or MAXTRANS parameters, the TABLESPACE clause, or the *storage_clause* with the CLUSTER clause.

Restriction: Object tables cannot be part of a cluster.

ON COMMIT — can be specified only if you are creating a temporary table. This clause specifies whether the data in the temporary table persists for the duration of a transaction or a session.

- DELETE ROWS — specifies that the temporary table is transaction specific (this is the default). Oracle will truncate the table (delete all its rows) after each commit.
- PRESERVE ROWS — specifies that the temporary table is session specific. Oracle will truncate the table (delete all its rows) when you terminate the session.

OID_clause — lets you specify whether the object identifier (OID) of the object table should be system generated or should be based on the primary key of the table. The default is SYSTEM GENERATED.

Restrictions:

- You cannot specify OBJECT IDENTIFIER IS PRIMARY KEY unless you have already specified a PRIMARY KEY constraint for the table.
- You cannot specify this clause for a nested table.

Note: A primary key OID is locally (but not necessarily globally) unique. If you require a globally unique identifier, you must ensure that the primary key is globally unique.

OID_index_clause — This clause is relevant only if you have specified the *OID_clause* as SYSTEM GENERATED. It specifies an index, and optionally its storage characteristics, on the hidden object identifier column.

- *index* — is the name of the index on the hidden system-generated object identifier column. If not specified, Oracle generates a name.

hash_partitioning_clause — specifies that the table is to be partitioned using the hash method. Oracle assigns rows to partitions using a hash function on values found in columns designated as the partitioning key. For more information on hash partitioning, see *Oracle8i Concepts*.

- *column_list* — is an ordered list of columns used to determine into which partition a row belongs (**the partitioning key**).

Restrictions:

- You cannot specify more than 16 columns in *column_list*.
- The *column_list* cannot contain the ROWID or UROWID pseudocolumns.
- The columns in *column_list* can be of any built-in datatype except ROWID, LONG, or LOB.

You can specify hash partitioning in one of two ways:

- You can specify the number of partitions. In this case, Oracle assigns partition names of the form SYS_P*nnn*. The STORE IN clause specifies one or more tablespaces where the hash partitions are to be stored. The number of tablespaces does not have to equal the number of partitions. If the number of partitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.
- Alternatively, you can specify individual partitions by name. The TABLESPACE clause specifies where the partition should be stored.

Note: The only attribute you can specify for hash partitions (or subpartitions) is TABLESPACE. Hash partitions inherit all other attributes from table-level defaults. Hash subpartitions inherit any attributes specified at the partition level, and inherit all other attributes from the table-level defaults.

Tablespace storage specified at the table level is overridden by tablespace storage specified at the partition level, which in turn is overridden by tablespace storage specified at the subpartition level.

range_partitioning_clause

PARTITION BY RANGE — specifies that the table is partitioned on ranges of values from *column_list*. For an index-organized table, *column_list* must be a subset of the primary key columns of the table.

column_list — is an ordered list of columns used to determine into which partition a row belongs (the **partitioning key**).

composite_partitioning_clause — specifies that table is to be first range partitioned, and then the partitions further partitioned into hash subpartitions. This combination of range partitioning and hash subpartitioning is called **composite partitioning**.

subpartition_clause — specifies that Oracle should subpartition by hash each partition in *table*. The subpartitioning *column_list* is unrelated to the partitioning key.

SUBPARTITIONS *quantity* — specifies the default number of subpartitions in each partition of *table*, and optionally one or more tablespaces in which they are to be stored.

The default value is 1. If you do not specify the *subpartition_clause* here, Oracle will create each partition with one hash subpartition unless you subsequently specify the *partition_level_hash_subpartitioning* clause.

partition_definition	PARTITION *partition*	specifies the physical partition attributes. If *partition* is omitted, Oracle generates a name with the form SYS_P*n* for the partition. The *partition* must conform to the rules for naming schema objects and their part as described in "Schema Object Naming Rules" on page 2-67.

Notes:

- You can specify up to 64K-1 partitions and 64K-1 subpartitions. For a discussion of factors that might impose practical limits less than this number, please refer to *Oracle8i Administrator's Guide*.
- You can create a partitioned table with just one partition. Note, however, that a partitioned table with one partition is different from a nonpartitioned table. For instance, you cannot add a partition to a nonpartitioned table.

VALUES LESS THAN	specifies the noninclusive upper bound for the current partition.
value_list	is an ordered list of literal values corresponding to *column_list* in the *partition_by_range_clause*. You can substitute the keyword MAXVALUE for any literal in *value_list*. MAXVALUE specifies a maximum value that will always sort higher than any other value, including NULL. Specifying a value other than MAXVALUE for the highest partition bound imposes an implicit integrity constraint on the table. See *Oracle8i Concepts* for more information about partition bounds.
	Note: If *table* is partitioned on a DATE column, and if the NLS date format does not specify the century with the year, you must use the TO_DATE function with a 4-character format mask for the year. The NLS date format is determined implicitly by NLS_TERRITORY or explicitly by NLS_DATE_FORMAT. For more information on these initialization parameters, see *Oracle8i National Language Support Guide*. See also "Partitioned Table Example" on page 7-389.
LOB_storage_clause	lets you specify LOB storage characteristics for one or more LOB items in this partition. If you do not specify the *LOB_storage_clause* for a LOB item, Oracle generates a name for each LOB data partition. The system-generated names for LOB data and LOB index partitions take the form SYS_LOB_P*n* and SYS_IL_P*n*, respectively, where P stands for "partition" and *n* is a system-generated number.
partition_level_ subpartitioning	lets you specify hash subpartitions for *partition*. This clause overrides the default settings established in the *subpartition_clause*. **Restriction:** You can specify this clause only for a composite-partitioned table.

- You can specify individual subpartitions by name, and optionally the tablespace where each should be stored, or
- You can specify the number of subpartitions (and optionally one or more tablespaces where they are to be stored). In this case, Oracle assigns subpartition names of the form SYS_SUBP*nnn*. The number of tablespaces does not have to equal the number of subpartitions. If the number of partitions is greater than the number of tablespaces, Oracle cycles through the names of the tablespaces.

row_movement_clause determines whether a row can be moved to a different partition or subpartition because of a change to one or more of its key values during an update operation.

Restriction: You can specify this clause only for a partitioned table.

ENABLE allows Oracle to move a row to a different partition or subpartition as the result of an update to the partitioning or subpartitioning key.

WARNING: Moving a row in the course of an UPDATE operation changes that row's ROWID.

DISABLE returns an error if an update to a partitioning or subpartitioning key would result in a row moving to a different partition or subpartition. This is the default.

parallel_clause causes creation of the table to be parallelized, and sets the default degree of parallelism for queries and DML on the table after creation.

NOPARALLEL specifies serial execution. This is the default.

PARALLEL causes Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

PARALLEL *integer* specifies the **degree of parallelism**, which is the number of parallel threads used in the parallel operation. Each parallel thread may use one or two parallel execution servers. Normally Oracle calculates the optimum degree of parallelism, so it is not necessary for you to specify *integer*.

Restriction: If *table* contains any columns of LOB or user-defined object type, this statement as well as subsequent INSERT, UPDATE, or DELETE operations on *table* are executed serially without notification. Subsequent queries, however, will be executed in parallel.

Notes

- This syntax supersedes syntax appearing in earlier releases of Oracle. Superseded syntax is still supported for backward compatibility, but may result in slightly different behavior. For more information, see *Oracle8i Migration*.

- A parallel hint overrides the effect of the *parallel_clause*.
- If the query portion of a parallel DML statement (INSERT, UPDATE, or DELETE) or a parallel DDL statement (CREATE TABLE ... AS SELECT) statement references a *remote object*, the operation is executed serially without notification.

For more information on parallelized operations, see *Oracle8i Tuning*, *Oracle8i Concepts*, and *Oracle8i Parallel Server Concepts and Administration*.

enable_disable_clause lets you specify whether Oracle should apply a constraint. By default, constraints are created in ENABLE VALIDATE state. For more information on constraints, see "constraint_clause" on page 7-217.

Restrictions:

- To enable or disable any integrity constraint, you must have defined the constraint in this or a previous statement.
- You cannot enable a referential integrity constraint unless the referenced unique or primary key constraint is already enabled.

ENABLE specifies that the constraint will be applied to all new data in the table.

- VALIDATE additionally specifies that all old data also complies with the constraint. An enabled validated constraint guarantees that all data is and will continue to be valid.

  If any row in the table violates the integrity constraint, the constraint remains disabled and Oracle returns an error. If all rows comply with the constraint, Oracle enables the constraint. Subsequently, if new data violates the constraint, Oracle does not execute the statement and returns an error indicating the integrity constraint violation.

  If you place a primary key constraint in ENABLE VALIDATE mode, the validation process will verify that the primary key columns contain no nulls. To avoid this overhead, mark each column in the primary key NOT NULL before enabling the table's primary key constraint. (For optimal results, do this before entering data into the column.)

- NOVALIDATE ensures that all new DML operations on the constrained data comply with the constraint. This clause does not ensure that existing data in the table complies with the constraint and therefore does not require a table lock.
- If you specify neither VALIDATE nor NOVALIDATE, the default is VALIDATE.
- If you enable a unique or primary key constraint, and if no index exists on the key, Oracle creates a unique index. This index is dropped if the constraint is subsequently disabled, so Oracle rebuilds the index every time the constraint is enabled.

  To avoid rebuilding the index and eliminate redundant indexes, create new primary key and unique constraints initially disabled. Then create (or use existing) nonunique indexes to enforce the constraint. Oracle does not drop a nonunique index when the constraint is disabled, so subsequent ENABLE operations are facilitated.

- If you change the state of any single constraint from ENABLE NOVALIDATE to ENABLE VALIDATE, the operation can be performed in parallel, and does not block reads, writes, or other DDL operations.

Restriction: You cannot enable a foreign key that references a unique or primary key that is disabled.

DISABLE — disables the integrity constraint. Disabled integrity constraints appear in the data dictionary along with enabled constraints. If you do not specify this clause when creating a constraint, Oracle automatically enables the constraint.

- DISABLE VALIDATE disables the constraint and drops the index on the constraint, but keeps the constraint valid. This feature is most useful in data warehousing situations, where the need arises to load into a range-partitioned table a quantity of data with a distinct range of values in the unique key. In such situations, the disable validate state enables you to save space by not having an index. You can then load data from a nonpartitioned table into a partitioned table using the *exchange_partition_clause* of the ALTER TABLE statement. All other modifications to the table by other SQL statements are disallowed.

  If the unique key coincides with the partitioning key of the partitioned table, disabling the constraint saves overhead and has no detrimental effects. If the unique key does not coincide with the partitioning key, Oracle performs automatic table scans during the exchange to validate the constraint, which might offset the benefit of loading without an index.

- DISABLE NOVALIDATE signifies that Oracle makes no effort to maintain the constraint (because it is disabled) and cannot guarantee that the constraint is true (because it is not being validated). For information on when to use this setting, see *Oracle8i Tuning*.

  You cannot drop a table whose primary key is being referenced by a foreign key even if the foreign key constraint is in DIASABLE NOVALIDATE state. Further, the optimizer can use constraints in DISABLE NOVALIDATE state.

- If you specify neither VALIDATE nor NOVALIDATE, the default is NOVALIDATE.
- If you disable a unique or primary key constraint that is using a unique index, Oracle drops the unique index.

UNIQUE — enables or disables the unique constraint defined on the specified column or combination of columns.

PRIMARY KEY — enables or disables the table's primary key constraint.

CONSTRAINT — enables or disables the integrity constraint named *constraint*.

using_index_clause — specifies parameters for the index Oracle creates to enforce a unique or primary key constraint. Oracle gives the index the same name as the constraint. You can choose the values of the INITRANS, MAXTRANS, TABLESPACE, STORAGE, and PCTFREE parameters for the index. These parameters are described earlier in this statement. For a description of NOSORT and of LOGGING | NOLOGGING in relation to indexes, see "CREATE INDEX" on page 7-273.

	Restriction: Use these parameters only when enabling unique and primary key constraints.
EXCEPTIONS INTO	specifies a table into which Oracle places information about rows that violate the integrity constraint. The table must exist on your local database before you use this clause. If you omit *schema*, Oracle assumes the exception table is in your own schema. **Note:** You must create an appropriate exceptions report table to accept information from the EXCEPTIONS INTO clause of the *enable_disable_clause* before enabling the constraint. You can create an exception table by submitting the script UTLEXCPT1.SQL, which creates a table named EXCEPTIONS. You can create additional exceptions tables with different names by modifying and resubmitting the script. (You can use the UTLEXCPT1.SQL script with index-organized tables. You could not use earlier versions of the script for this purpose. See *Oracle8i Migration* for compatibility information.) For more information on identifying exceptions, see *Oracle8i Application Developer's Guide - Fundamentals*.
CASCADE	disables any integrity constraints that depend on the specified integrity constraint. To disable a primary or unique key that is part of a referential integrity constraint, you must specify this clause. **Restriction:** You can specify CASCADE only if you have specified DISABLE.
CACHE	for data that will be accessed frequently, specifies that the blocks retrieved for this table are placed at the most recently used end of the LRU list in the buffer cache when a full table scan is performed. This clause is useful for small lookup tables. As a parameter in the *LOB_storage_clause*, CACHE specifies that Oracle preallocates and retains LOB data values in memory for faster access. **Restriction:** You cannot specify CACHE for an index-organized table.
NOCACHE	for data that will not be accessed frequently, specifies that the blocks retrieved for this table are placed at the least recently used end of the LRU list in the buffer cache when a full table scan is performed. This is the default. For LOBs, the LOB value either is not brought into the buffer cache or is brought into the buffer cache and placed at the least recently used end of the LRU list. As a parameter in the *LOB_storage_clause*, NOCACHE specifies that LOB values are not preallocated in memory. **Restriction:** You cannot specify NOCACHE for an index-organized table.
MONITORING	specifies that modification statistics can be collected on this table. These statistics are estimates of the number of rows affected by DML statements over a particular period of time. They are available for use by the optimizer or for analysis by the user. **Restriction:** You cannot specify MONITORING for a temporary table.
NOMONITORING	specifies that the table will not have modification statistics collected. This is the default. **Restriction:** You cannot specify NOMONITORING for a temporary table.

AS *subquery* inserts the rows returned by the subquery into the table upon its creation. See "SELECT and Subqueries" on page 7-541.

Restrictions:

- The number of columns in the table must equal the number of expressions in the subquery.
- The column definitions can specify only column names, default values, and integrity constraints, not datatypes.
- You cannot define a referential integrity constraint in a CREATE TABLE statement that contains AS *subquery*. Instead, you must create the table without the constraint and then add it later with an ALTER TABLE statement.

If you specify the *parallel_clause* in this statement, Oracle will ignore any value you specify for the INITIAL storage parameter, and will instead use the value of the NEXT parameter. For information on these parameters, see the "storage_clause" on page 7-575.

Oracle derives datatypes and lengths from the subquery. Oracle also follows the following rules for integrity constraints:

- Oracle automatically defines any NOT NULL constraints on columns in the new table that existed on the corresponding columns of the selected table if the subquery selects the column rather than an expression containing the column.
- If a CREATE TABLE statement contains both AS *subquery* and a CONSTRAINT clause or an ENABLE clause with the EXCEPTIONS INTO clause, Oracle ignores AS *subquery*. If any rows violate the constraint, Oracle does not create the table and returns an error.

If all expressions in *subquery* are columns, rather than expressions, you can omit the columns from the table definition entirely. In this case, the names of the columns of table are the same as the columns in *subquery*.

You can use *subquery* in combination with the TO_LOB function to convert the values in a LONG column in another table to LOB values in a column of the table you are creating. For a discussion of why and when to copy LONGs to LOBs, see *Oracle8i Migration*. For a description of how to use the TO_LOB function, see "Conversion Functions" on page 4-4.

Note: If *subquery* returns (in part or totally) the equivalent of an existing materialized view, Oracle may use the materialized view (for query rewrite) in place of one or more tables specified in *subquery*. For more information on materialized views and query rewrite, see *Oracle8i Tuning*.

---

order_by_clause orders rows returned by the statements. For more information on the *order_by_clause*, refer to "SELECT and Subqueries" on page 7-541.

Note: When specified with CREATE TABLE, this clause does not necessarily order data cross the entire table. (For example, it does not order across partitions.) Specify this clause if you intend to create an index on the same key as the ORDER BY key column. Oracle will cluster data on the ORDER BY key so that it corresponds to the index key.

For object tables, *subquery* can contain either one expression corresponding to the table type, or the number of top-level attributes of the table type.

Examples

General Example To define the EMP table owned by SCOTT, you could issue the following statement:

```
CREATE TABLE scott.emp
   (empno     NUMBER        CONSTRAINT pk_emp PRIMARY KEY,
    ename     VARCHAR2(10)  CONSTRAINT nn_ename NOT NULL
                            CONSTRAINT upper_ename
   CHECK (ename = UPPER(ename)),
       job        VARCHAR2(9),
       mgr        NUMBER       CONSTRAINT fk_mgr
                               REFERENCES scott.emp(empno),
       hiredate  DATE          DEFAULT SYSDATE,
       sal       NUMBER(10,2)  CONSTRAINT ck_sal
   CHECK (sal > 500),
          comm      NUMBER(9,0)   DEFAULT NULL,
          deptno    NUMBER(2)     CONSTRAINT nn_deptno NOT NULL
                                  CONSTRAINT fk_deptno
                                  REFERENCES scott.dept(deptno) )
   PCTFREE 5 PCTUSED 75;
```

This table contains eight columns. The EMPNO column is of datatype NUMBER and has an associated integrity constraint named PK_EMP. The HIRDEDATE column is of datatype DATE and has a default value of SYSDATE, and so on.

This table definition specifies a PCTFREE of 5 and a PCTUSED of 75, which is appropriate for a relatively static table. The definition also defines integrity constraints on some columns of the EMP table.

Temporary Table Example The following statement creates a temporary table FLIGHT_SCHEDULE for use in an automated airline reservation scheduling system. Each client has its own session and can store temporary schedules. The temporary schedules are deleted at the end of the session.

```
CREATE GLOBAL TEMPORARY TABLE flight_schedule (
   startdate DATE,
   enddate DATE,
   cost NUMBER)
   ON COMMIT PRESERVE ROWS;
```

Storage Example To define the sample table SALGRADE in the HUMAN_RESOURCE tablespace with a small storage capacity and limited allocation potential, issue the following statement:

```
CREATE TABLE salgrade
   ( grade  NUMBER  CONSTRAINT pk_salgrade
                    PRIMARY KEY
                    USING INDEX TABLESPACE users_a,
     losal  NUMBER,
     hisal  NUMBER )
   TABLESPACE human_resource
   STORAGE (INITIAL     6144
         NEXT        6144
         MINEXTENTS     1
         MAXEXTENTS     5 );
```

The above statement also defines a primary key constraint on the GRADE column and specifies that the index Oracle creates to enforce this constraint is created in the USERS_A tablespace.

For more examples of defining integrity constraints, see the "constraint_clause" on page 7-217.

PARALLEL Example The following statement creates a table using an optimum number of parallel execution servers to scan SCOTT.EMP and to populate EMP_DEPT:

```
CREATE TABLE emp_dept
   PARALLEL
   AS SELECT * FROM scott.emp
   WHERE deptno = 10;
```

Using parallelism speeds up the creation of the table because Oracle uses parallel execution servers to create the table. After the table is created, querying the table is also faster, because the same degree of parallelism is used to access the table.

NOPARALLEL Example The following statement creates a table serially. Subsequent DML and queries on the table will also be serially executed.

```
CREATE TABLE emp_dept
   AS SELECT * FROM scott.emp
   WHERE deptno = 10;
```

ENABLE VALIDATE Example The following statement creates the DEPT table, defines a primary key constraint, and places it in ENABLE VALIDATE state:

```
CREATE TABLE dept
   (deptno  NUMBER (2) PRIMARY KEY,
    dname   VARCHAR2(10),
    loc     VARCHAR2(9) )
   TABLESPACE user_a;
```

DISABLE Example The following statement creates the DEPT table and defines a disabled primary key constraint:

```
CREATE TABLE dept
   (deptno   NUMBER (2)   PRIMARY KEY DISABLE,
    dname    VARCHAR2(10),
    loc      VARCHAR2(9) );
```

EXCEPTIONS INTO Example The following example creates the ORDER_EXEPTIONS table to hold rows from an index-organized table ORDERS that violate integrity constraint CHECK_ORDERS:

```
CREATE TABLE orders
   (ord_num  NUMBER PRIMARY KEY,
    ord_quantity NUMBER)
   ORGANIZATION INDEX;

EXECUTE DBMS_IOT.BUILD_EXCEPTIONS_TABLE
   ('SCOTT', 'ORDERS', 'ORDER_EXCEPTIONS');

ALTER TABLE orders
   ADD CONSTRAINT CHECK_ORDERS CHECK (ord_quantity > 0)
   EXCEPTIONS INTO ORDER_EXCEPTIONS;
```

To specify an exception table, you must have the privileges necessary to insert rows into the table. For more information, see "INSERT" on page 7-512. To examine the identified exceptions, you must have the privileges necessary to query the exceptions table. For information on these privileges, see "SELECT and Subqueries" on page 7-541.

Nested Table Example The following statement creates relational table EMPLOYEE with a nested table column PROJECTS:

```
CREATE TABLE employee
   (empno NUMBER, name CHAR(31), projects PROJ_TABLE_TYPE)
   NESTED TABLE projects STORE AS nested_proj_table(
      (PRIMARY KEY (nested_table_id, pno)) ORGANIZATION INDEX)
   RETURN AS LOCATOR;
```

LOB Column Example The following statement creates table LOB_TAB with two LOB columns and specifies the LOB storage characteristics:

```
CREATE TABLE lob_tab (col1 BLOB, col2 CLOB)
   STORAGE (INITIAL 256 NEXT 256)
   LOB (col1, col2) STORE AS
      (TABLESPACE lob_seg_ts
       STORAGE (INITIAL 6144 NEXT 6144)
       CHUNK 4000
       NOCACHE LOGGING);
```

In the example, Oracle rounds the CHUNK up to 4096 (the nearest multiple of the block size of 2048).

Index-Organized Table Example The following statement creates an index-organized table:

```
CREATE TABLE docindex
  ( token                 CHAR(20),
    doc_oid               INTEGER,
    token_frequency       SMALLINT,
    token_occurrence_data  VARCHAR2(512),
       CONSTRAINT pk_docindex PRIMARY KEY (token, doc_oid) )
  ORGANIZATION INDEX TABLESPACE text_collection
  PCTTHRESHOLD 20 INCLUDING token_frequency
  OVERFLOW TABLESPACE text_collection_overflow;
```

Partitioned Table Example The following statement creates a table with three partitions:

```
CREATE TABLE stock_xactions
      (stock_symbol CHAR(5),
       stock_series CHAR(1),
       num_shares NUMBER(10),
       price NUMBER(5,2),
       trade_date DATE)
   STORAGE (INITIAL 100K NEXT 50K) LOGGING
   PARTITION BY RANGE (trade_date)
     (PARTITION sx1992 VALUES LESS THAN (TO_DATE('01-JAN-1993','DD-MON-YYYY'))
        TABLESPACE ts0 NOLOGGING,
      PARTITION sx1993 VALUES LESS THAN (TO_DATE('01-JAN-1994','DD-MON-YYYY'))
        TABLESPACE ts1,
      PARTITION sx1994 VALUES LESS THAN (TO_DATE('01-JAN-1995','DD-MON-YYYY'))
        TABLESPACE ts2);
```

For information about partitioned table maintenance operations, see the *Oracle8i Administrator's Guide.*

Partitioned Table with LOB Columns Example This statement creates a partitioned table PT with two partitions P1 and P2, and three LOB columns, B, C, and D:

```
CREATE TABLE PT (A NUMBER, B BLOB, C CLOB, D CLOB)
   LOB (B,C,D) STORE AS (STORAGE (NEXT 20M))
   PARTITION BY RANGE (A)
   (PARTITION P1 VALUES LESS THAN (10) TABLESPACE TS1
      LOB (B,D) STORE AS (TABLESPACE TSA STORAGE (INITIAL 20M)),
   PARTITION P2 VALUES LESS THAN (20)
      LOB (B,C) STORE AS (TABLESPACE TSB)
   TABLESPACE TSX;
```

Partition P1 will be in tablespace TS1. The LOB data partitions for B and D will be in tablespace TSA. The LOB data partition for C will be in tablespace TS1. The storage attribute INITIAL is specified for LOB columns B and D; other attributes will be inherited from the default table-level specification. The default LOB storage attributes not specified at the table level will be inherited from the tablespace TSA for columns B and D and tablespace TS1 for column C. LOB index partitions will be in the same tablespaces as the corresponding LOB data partitions. Other storage attributes will be based on values of the corresponding attributes of the LOB data partitions and default attributes of the tablespace where the index partitions reside.

Partition P2 will be in the default tablespace TSX. The LOB data for B and C will be in tablespace TSB. The LOB data for D will be in tablespace TSX. The LOB index for columns B and C will be in tablespace TSB. The LOB index for column D will be in tablespace TSX.

Hash-Partitioned Table Example This statement creates a table partitioned by hash on columns containing data about chemicals. The hash partitions are stored in tablespaces TBS1, TBS2, TBS3, and TBS4:

```
CREATE TABLE exp_data (
      d DATE, temperature NUMBER, Fe2O3_concentration NUMBER,
      HCl_concentration NUMBER, Au_concentration NUMBER,
      amps NUMBER, observation VARCHAR(4000))
   PARTITION BY HASH (HCl_concentration, Au_concentration)
   PARTITIONS 32 STORE IN (tbs1, tbs2, tbs3, tbs4);
```

Composite-Partitioned Table Example This statement creates a composite-partitioned table. The range partitioning facilitates data and partition pruning by

sale date. The hash subpartitioning enables subpartition elimination for queries by a specific item number. Most of the partitions consist of 8 subpartitions. However, the partition covering the slowest quarter will have 4 subpartitions, and the partition covering the busiest quarter will have 16 subpartitions.

```
CREATE TABLE sales (item INTEGER, qty INTEGER,
                    store VARCHAR(30),
                    dept NUMBER, sale_date DATE)
   PARTITION BY RANGE (sale_date)
   SUBPARTITION BY HASH(item)
   SUBPARTITIONS 8
   STORE IN (tbs1, tbs2, tbs3, tbs4, tbs5, tbs6, tbs7, tbs8)
   (PARTITION q1_1997
      VALUES LESS THAN (TO_DATE('01-apr-1997', 'dd-mon-yyyy')),
    PARTITION q2_1997
      VALUES LESS THAN (TO_DATE('01-jul-1997', 'dd-mon-yyyy')),
    PARTITION q3_1997
      VALUES LESS THAN (TO_DATE('01-oct-1997', 'dd-mon-yyyy'))
      (SUBPARTITION q3_1997_s1 TABLESPACE ts1,
       SUBPARTITION q3_1997_s2 TABLESPACE ts3,
       SUBPARTITION q3_1997_s3 TABLESPACE ts5,
       SUBPARTITION q3_1997_s4 TABLESPACE ts7),
    PARTITION q4_1997
      VALUES LESS THAN (TO_DATE('01-jan-1998', 'dd-mon-yyyy'))
      SUBPARTITIONS 16
      STORE IN (tbs1, tbs3, tbs5, tbs7, tbs8, tbs9, tbs10,
                tbs11),
    PARTITION q1_1998
      VALUES LESS THAN (TO_DATE('01-apr-1998', 'dd-mon-yyyy')));
```

Object Table Examples Consider object type DEPT_T:

```
CREATE TYPE dept_t AS OBJECT
   ( dname VARCHAR2(100),
     address VARCHAR2(200) );
```

Object table DEPT holds department objects of type DEPT_T:

```
CREATE TABLE dept OF dept_t;
```

The following statement creates object table SALESREPS with a user-defined object type, SALESREP_T:

```
CREATE OR REPLACE TYPE salesrep_t AS OBJECT
  ( repId NUMBER,
    repName VARCHAR2(64));
```

```
CREATE TABLE salesreps OF salesrep_t;
```

Nested Table Example The following statement creates relational table EMPLOYEE with a nested table column PROJECTS:

```
CREATE TABLE employee (empno NUMBER, name CHAR(31),
   projects PROJ_TABLE_TYPE)
   NESTED TABLE projects STORE AS nested_proj_table;
```

REF Example The following example creates object type DEPT_T and object table DEPT to store instances of all departments. A table with a scoped REF is then created.

```
CREATE TYPE dept_t AS OBJECT
   ( dname   VARCHAR2(100),
     address VARCHAR2(200) );

CREATE TABLE dept OF dept_t;

CREATE TABLE emp
   ( ename   VARCHAR2(100),
     enumber NUMBER,
     edept   REF dept_t SCOPE IS dept );
```

The following statement creates a table with a REF column which has a referential constraint defined on it:

```
CREATE TABLE emp
   ( ename   VARCHAR2(100),
     enumber NUMBER,
     edept   REF dept_t REFERENCES dept);
```

User-Defined OID Example This example creates an object type and a corresponding object table whose OID is primary key based:

```
CREATE TYPE emp_t AS OBJECT (empno NUMBER, address CHAR(30));
CREATE TABLE emp OF emp_t (empno PRIMARY KEY)
   OBJECT IDENTIFIER IS PRIMARY KEY;
```

You can subsequently reference the EMP object table in either of the following two ways:

```
CREATE TABLE dept (dno NUMBER
                   mgr_ref REF emp_t SCOPE IS emp);
CREATE TABLE dept (
    dno NUMBER,
```

```
    mgr_ref REF emp_t CONSTRAINT mgr_in_emp REFERENCES emp);
```

Constraints on Type Columns Example

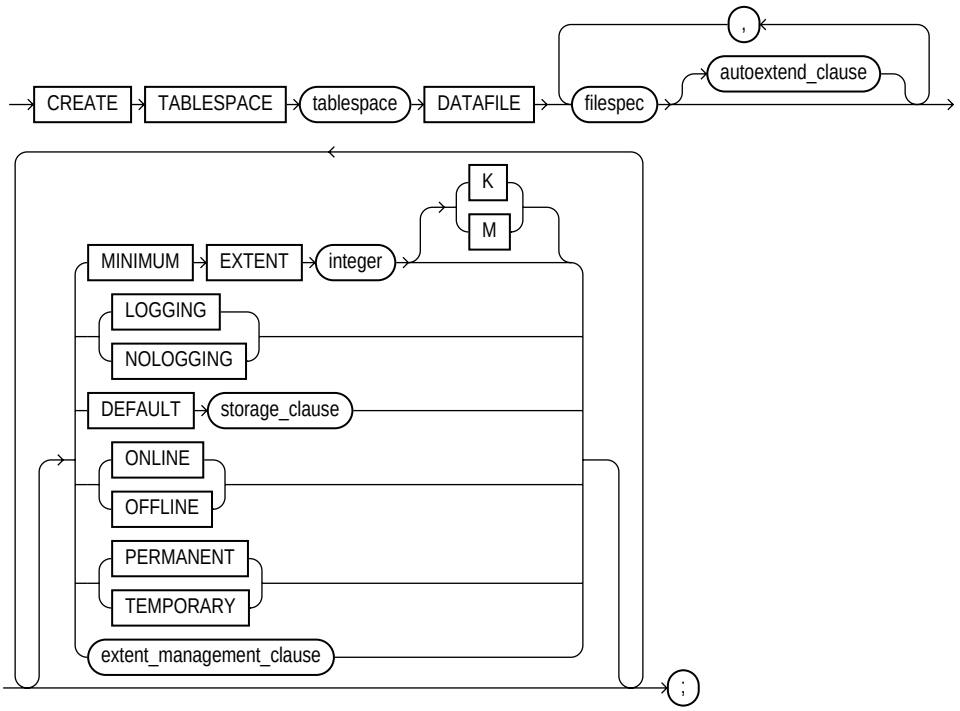
```
CREATE TYPE address AS OBJECT
  ( hno NUMBER,
    street VARCHAR2(40),
    city VARCHAR2(20),
    zip VARCHAR2(5),
    phone VARCHAR2(10) );

CREATE TYPE person AS OBJECT
  ( name VARCHAR2(40),
    dateofbirth DATE,
    homeaddress address,
    manager REF person );

CREATE TABLE persons OF person
  ( homeaddress NOT NULL
      UNIQUE (homeaddress.phone),
      CHECK (homeaddress.zip IS NOT NULL),
      CHECK (homeaddress.city <> 'San Francisco') );
```

PARALLEL Example The following statement creates a table using 10 parallel execution servers, 5 to scan SCOTT.EMP and another 5 to populate EMP_DEPT:

```
CREATE TABLE emp_dept
   PARALLEL (5)
   AS SELECT * FROM scott.emp
   WHERE deptno = 10;
```

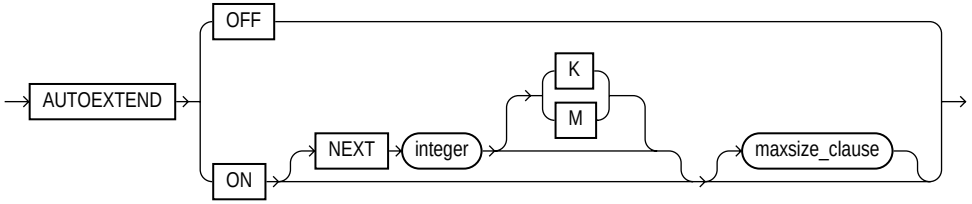
CREATE TABLESPACE

Syntax

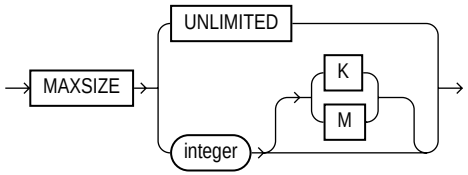


filespec: See "filespec" on page 7-490.

autoextend_clause::=

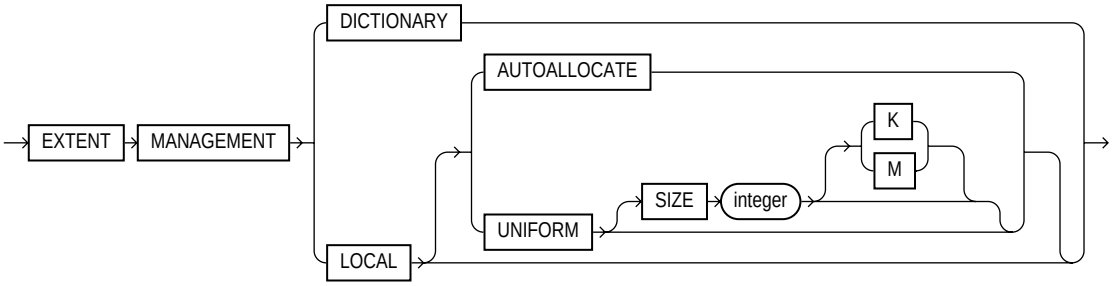


maxsize_clause::=



storage_clause: See "storage_clause" on page 7-575.

extent_management_clause::=



Purpose

To create a tablespace. A **tablespace** is an allocation of space in the database that can contain schema objects. For information on tablespaces, see *Oracle8i Concepts*.

When you create a tablespace, it is initially a read-write tablespace. You can subsequently use the ALTER TABLESPACE statement to take the tablespace offline or online, add datafiles to it, or make it a read-only tablespace. See "ALTER TABLESPACE" on page 7-164.

You can also drop a tablespace from the database with the DROP TABLESPACE statement. See "DROP TABLESPACE" on page 7-477.

Prerequisites

You must have CREATE TABLESPACE system privilege. Also, the SYSTEM tablespace must contain at least two rollback segments including the SYSTEM rollback segment.

Before you can create a tablespace you must create a database to contain it. See "CREATE DATABASE" on page 7-249.

Keywords and Parameters

tablespace is the name of the tablespace to be created.

DATAFILE *filespec* specifies the datafile or files to make up the tablespace. See "filespec" on page 7-490.

Note: For operating systems that support raw devices, the *filespec* REUSE keyword has no meaning when specifying a raw device as a datafile. Such a CREATE TABLESPACE statement will succeed whether or not you specify REUSE.

autoextend_clause enables or disables the automatic extension of the datafile.

OFF	disables autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER TABLESPACE AUTOEXTEND statements.
ON	enables autoextend.
NEXT	specifies the disk space to allocate to the datafile when more extents are required.
maxsize_clause	specifies the maximum disk space allowed for allocation to the datafile.
UNLIMITED	sets no limit on allocating disk space to the datafile.

MINIMUM EXTENT *integer* controls free space fragmentation in the tablespace by ensuring that every used or free extent size in a tablespace is at least as large as, and is a multiple of, *integer*. For more information about using MINIMUM EXTENT to control fragmentation, see *Oracle8i Concepts*.

Note: This clause is not relevant for a dictionary-managed temporary tablespace.

LOGGING | NOLOGGING specifies the default logging attributes of all tables, indexes, and partitions within the tablespace. LOGGING is the default.

The tablespace-level logging attribute can be overridden by logging specifications at the table, index, and partition levels.

Only the following operations support the NOLOGGING mode:

DML: direct-load INSERT (serial or parallel), Direct Loader (SQL*Loader)

DDL: CREATE TABLE ... AS SELECT, CREATE INDEX, ALTER INDEX ... REBUILD, ALTER INDEX ... REBUILD PARTITION, ALTER INDEX ... SPLIT PARTITION, ALTER TABLE ... SPLIT PARTITION, and ALTER TABLE ... MOVE PARTITION

In NOLOGGING mode, data is modified with minimal logging (to mark new extents INVALID and to record dictionary changes). When applied during media recovery, the extent invalidation records mark a range of blocks as logically corrupt, because the redo data is not logged. Therefore, if you cannot afford to lose the object, you should take a backup after the NOLOGGING operation.

DEFAULT *storage_clause*
: specifies the default storage parameters for all objects created in the tablespace. For a dictionary-managed temporary tablespace, Oracle considers only the NEXT parameter of the *storage_clause*. For information on storage parameters, see the "storage_clause" on page 7-575.

ONLINE
: makes the tablespace available immediately after creation to users who have been granted access to the tablespace. This is the default.

OFFLINE
: makes the tablespace unavailable immediately after creation.

  The data dictionary view DBA_TABLESPACES indicates whether each tablespace is online or offline.

PERMANENT
: specifies that the tablespace will be used to hold permanent objects. This is the default.

TEMPORARY
: specifies that the tablespace will be used only to hold temporary objects, for example, segments used by implicit sorts to handle ORDER BY clauses.

extent_management_clause
: specifies how the extents of the tablespace will be managed.

  DICTIONARY
  : specifies that the tablespace is managed using dictionary tables. This is the default

  LOCAL
  : specifies that tablespace is locally managed. Locally managed tablespaces have some part of the tablespace set aside for a bitmap. For a discussion of locally managed tablespaces, see *Oracle8i Concepts*.

    - AUTOALLOCATE specifies that the tablespace is system managed. Users cannot specify an extent size.
    - UNIFORM specifies that the tablespace is managed with uniform extents of SIZE bytes. Use K or M to specify the extent size in kilobytes or megabytes. The default SIZE is 1 megabyte.

    If you do not specify either AUTOALLOCATE or UNIFORM, then AUTOALLOCATE is the default.

    Restriction: If you specify LOCAL, you cannot specify DEFAULT *storage_clause*, MINIMUM EXTENT, or TEMPORARY.

Examples

DEFAULT Storage Example This statement creates a tablespace named TABSPACE_2 with one datafile:

```
CREATE TABLESPACE tabspace_2
   DATAFILE 'diska:tabspace_file2.dat' SIZE 20M
   DEFAULT STORAGE (INITIAL 10K NEXT 50K
                    MINEXTENTS 1 MAXEXTENTS 999)
   ONLINE;
```

AUTOEXTEND Example This statement creates a tablespace named TABSPACE_3 with one datafile. When more space is required, 50 kilobyte extents will be added up to a maximum size of 10 megabytes:

```
CREATE TABLESPACE tabspace_5
   DATAFILE 'diskb:tabspace_file3.dat' SIZE 500K REUSE
   AUTOEXTEND ON NEXT 500K MAXSIZE 10M;
```

MINIMUM EXTENT Example This statement creates tablespace TABSPACE_5 with one datafile and allocates every extent as a multiple of 64K:

```
CREATE TABLESPACE tabspace_3
   DATAFILE 'tabspace_file5.dbf' SIZE 2M
   MINIMUM EXTENT 64K
   DEFAULT STORAGE (INITIAL 128K NEXT 128K)
   LOGGING;
```

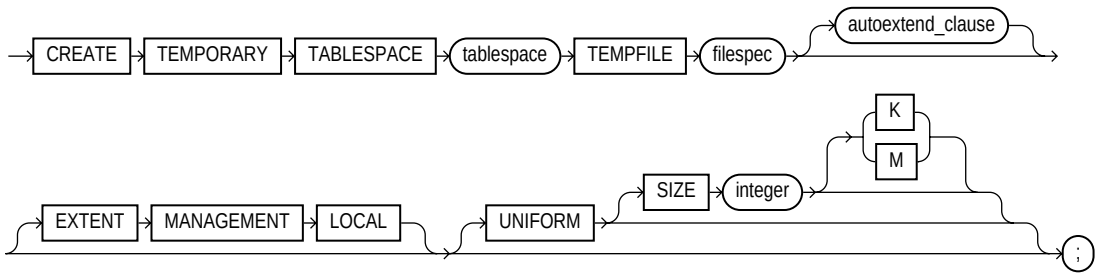
Locally Managed Example In the following statement, we assume that the database block size is 2K.

```
CREATE TABLESPACE tbs_1 DATAFILE 'file_1.f' SIZE 10M
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 128K;
```

This statement creates a locally managed tablespace in which every extent is 128K and each bit in the bit map describes 64 blocks.

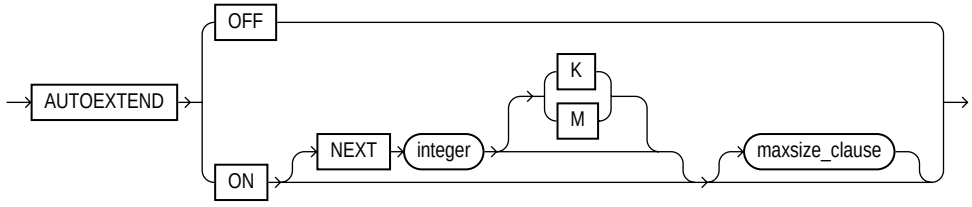
CREATE TEMPORARY TABLESPACE

Syntax

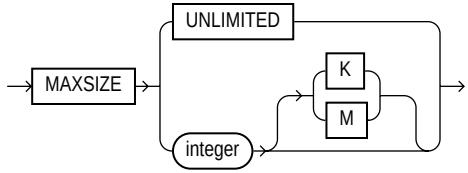


filespec: See "filespec" on page 7-490.

autoextend_clause::=



maxsize_clause::=



Purpose

To create a **temporary tablespace**, which is an allocation of space in the database that can contain schema objects for the duration of a session.

Prerequisites

You must have the CREATE TABLESPACE system privilege.

Keywords and Parameters

tablespace	is the name of the temporary tablespace.	
TEMPFILE *filespec*	specifies the tempfiles that make up the tablespace. See "filespec" on page 7-490.	
	Note: Media recovery does not recognize tempfiles.	
autoextend_clause	enables or disables the automatic extension of the tempfile.	
	OFF	disables autoextend if it is turned on. NEXT and MAXSIZE are set to zero. Values for NEXT and MAXSIZE must be respecified in further ALTER TABLESPACE AUTOEXTEND statements.
	ON	enables autoextend.
	NEXT	specifies the disk space to allocate to the tempfile when more extents are required.
	maxsize_clause	specifies the maximum disk space allowed for allocation to the tempfile.
	integer	specifies in bytes the maximum disk space allowed for allocation to the tempfile. Use K or M to specify this space in kilobytes or megabytes.
	UNLIMITED	sets no limit on allocating disk space to the tempfile.
EXTENT MANAGEMENT LOCAL	specifies that the tablespace is locally managed, meaning that some part of the tablespace is set aside for a bitmap. For a discussion of locally managed tablespaces, see *Oracle8i Concepts*.	
	UNIFORM	determines the size of the extents of the temporary tablespace in bytes. All extents of temporary tablespaces are the same size (uniform). If you do not specify this clause, Oracle uses uniform extents of 1M.
	SIZE	specifies in bytes the size of the tablespace extents. Use K or M to specify the size in kilobytes or megabytes. If you do not specify SIZE, Oracle uses the default extent size of 1 M.

Example

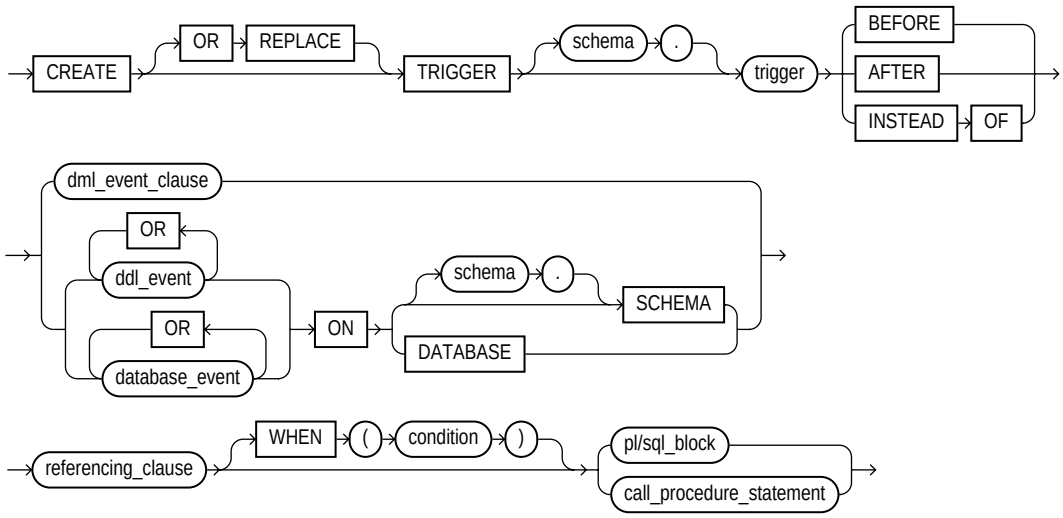
This statement creates a temporary tablespace in which each extent is 16 M.

```
CREATE TEMPORARY TABLESPACE tbs_1 TEMPFILE 'file_1.f'
   EXTENT MANAGEMENT LOCAL UNIFORM SIZE 16M;
```

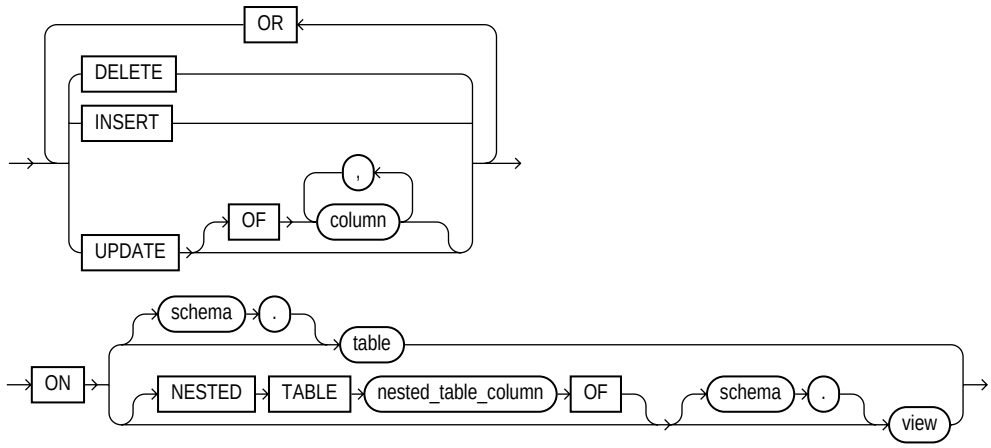
If we assume the default database block size of 2K, and that each bit in the map represents one extent, then each bit maps 8,000 blocks.

CREATE TRIGGER

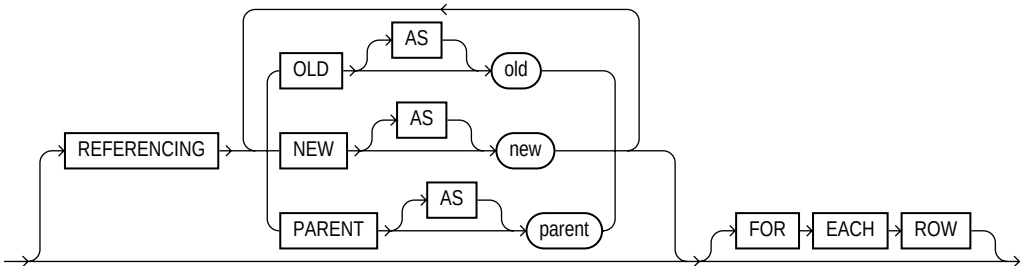
Syntax



dml_event_clause::=



referencing_clause::=



Purpose

To create and enable a database trigger. A **database trigger** is

- A stored PL/SQL block associated with a table, a schema, or the database
- An anonymous PL/SQL block or a call to a procedure implemented in PL/SQL or Java

Oracle automatically executes a trigger when specified conditions occur. For a description of the various types of triggers, see also *Oracle8i Concepts*.

For more information on how to design triggers for the above purposes, see *Oracle8i Application Developer's Guide - Fundamentals.*

Prerequisites

Before a trigger can be created, the user SYS must run the SQL script DBMSSTDX.SQL. The exact name and location of this script depend on your operating system.

- To create a trigger in your own schema on a table in your own schema or on your own schema (SCHEMA), you must have the CREATE TRIGGER privilege.
- To create a trigger in any schema on a table in any schema, or on another user's schema (*schema*.SCHEMA), you must have the CREATE ANY TRIGGER privilege.
- In addition to the preceding privileges, to create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER system privilege.

If the trigger issues SQL statements or calls procedures or functions, then the owner of the trigger must have the privileges necessary to perform these

operations. These privileges must be granted directly to the owner, rather than acquired through roles.

Keywords and Parameters

CREATE — creates a new trigger. When you create a trigger, Oracle enables it automatically. You can subsequently disable and enable a trigger with the DISABLE and ENABLE clause of the ALTER TRIGGER or ALTER TABLE statement. For information on how to enable and disable triggers, see "ALTER TRIGGER" on page 7-171 and "ALTER TABLE" on page 7-113.

If a trigger produces compilation errors, it is still created, but it fails on execution. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS. This means it effectively blocks all triggering DML statements until it is disabled, replaced by a version without compilation errors, or dropped.

OR REPLACE — re-creates the trigger if it already exists. Use this clause to change the definition of an existing trigger without first dropping it.

schema — is the schema to contain the trigger. If you omit *schema*, Oracle creates the trigger in your own schema.

trigger — is the name of the trigger to be created.

BEFORE — causes Oracle to fire the trigger before executing the triggering event. For row triggers, this is a separate firing before each affected row is changed.

Restrictions:

- You cannot specify a BEFORE trigger on a view or an object view.
- When defining a BEFORE trigger for LOB columns, you can read the :OLD value but not the :NEW value. You cannot write either the :OLD or the :NEW value.

AFTER — causes Oracle to fire the trigger after executing the triggering event. For row triggers, this is a separate firing after each affected row is changed.

Restrictions:

- You cannot specify an AFTER trigger on a view or an object view.
- When defining an AFTER trigger for LOB columns, you can read the :OLD value but not the :NEW value. You cannot write either the :OLD or the :NEW value.

Note: When you create a snapshot log for a table, Oracle implicitly creates an AFTER ROW trigger on the table. This trigger inserts a row into the snapshot log whenever an INSERT, UPDATE, or DELETE statement modifies the table's data. You cannot control the order in which multiple row triggers fire. Therefore, you should not write triggers intended to affect the content of the snapshot. For more information on snapshot logs, see CREATE MATERIALIZED VIEW LOG / SNAPSHOT LOG on page 7-314.

INSTEAD OF causes Oracle to fire the trigger instead of executing the triggering event. By default, INSTEAD OF triggers are activated for each row.

If a view is inherently updatable and has INSTEAD OF triggers, the triggers take preference. In other words, Oracle fires the triggers instead of performing DML on the view.

Restrictions:

- INSTEAD OF is a valid clause only for views. You cannot specify an INSTEAD OF trigger on a table.
- If a view has INSTEAD OF triggers, any views created on it must have INSTEAD OF triggers, even if the views are inherently updatable.
- When defining INSTEAD OF TRIGGERS for LOB columns, you can read both the :OLD and the :NEW value, but you cannot write either the :OLD or the :NEW values.

Note: You can create multiple triggers of the same type (BEFORE, AFTER, or INSTEAD OF) that fire for the same statement on the same table. The order in which Oracle fires these triggers is indeterminate. If your application requires that one trigger be fired before another of the same type for the same statement, combine these triggers into a single trigger whose trigger action performs the trigger actions of the original triggers in the appropriate order.

dml_event_clause specifies one of three DML statements that can cause the trigger to fire. Oracle fires the trigger in the existing user transaction.

DELETE causes Oracle to fire the trigger whenever a DELETE statement removes a row from the table or an element from a nested table.

INSTEAD causes Oracle to fire the trigger whenever an INSERT statement adds a row to table or an element to a nested table.

UPDATE causes Oracle to fire the trigger whenever an UPDATE statement changes a value in one of the columns specified after OF. If you omit OF, Oracle fires the trigger whenever an UPDATE statement changes a value in any column of the table or nested table.

For an UPDATE trigger, you can specify object type, varray, and REF columns after OF to indicate that the trigger should be fired whenever an UPDATE statement changes a value in one of the columns. However, you cannot change the values of these columns in the body of the trigger itself.

Note: Using OCI functions or the DBMS_LOB package to update LOB values or LOB attributes of object columns does not cause Oracle to fire triggers defined on the table containing the columns or the attributes.

Restrictions:

- You cannot specify OF with UPDATE for an INSTEAD OF trigger. Oracle fires INSTEAD OF triggers whenever an UPDATE changes a value in any column of the view.
- You cannot specify nested table or LOB columns with OF.
- See AS *subquery* of "CREATE VIEW" on page 7-430 for a list of constructs that prevent inserts, updates, or deletes on a view.

Performing DML operations directly on nested table columns does not cause Oracle to fire triggers defined on the table containing the nested table column

ddl_event is one of three DDL statements that can cause the trigger to fire. You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. You can create BEFORE and AFTER triggers for these events. Oracle fires the trigger in the existing user transaction.

CREATE	causes Oracle to fire the trigger whenever a CREATE statement adds a new database object to the data dictionary.
ALTER	causes Oracle to fire the trigger whenever an ALTER statement modifies a database object in the data dictionary.
DROP	causes Oracle to fire the trigger whenever a DROP statement removes a database object from the data dictionary.

Restriction: DDL triggers are supported only for the following database objects: cluster, function, index, package, procedure, role, sequence, synonym, table, tablespace, trigger, type, view, and user.

database_event describes a particular state of the database that can cause the trigger to fire. You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. For each of these triggering events, Oracle opens an autonomous transaction scope, fires the trigger, and commits any separate transaction (regardless of any existing user transaction). For more information on autonomous transaction scope, see *PL/SQL User's Guide and Reference*.

SERVERERROR	causes Oracle to fire the trigger whenever a server error message is logged.
LOGON	causes Oracle to fire the trigger whenever a client application logs onto the database.
LOGOFF	causes Oracle to fire the trigger whenever a client applications logs off the database.
STARTUP	causes Oracle to fire the trigger whenever the database is opened.
SHUTDOWN	causes Oracle to fire the trigger whenever an instance of the database is shut down.

Notes:

- Only AFTER triggers are relevant for LOGON, STARTUP, and SERVERERROR.
- Only BEFORE triggers are relevant for LOGOFF and SHUTDOWN.
- AFTER STARTUP and BEFORE SHUTDOWN triggers apply only to DATABASE.

ON — determines the database object on which the trigger is to be created.

[schema.] table | view — specifies the *schema* and *table* or *view* name of the of one of the following on which the trigger is to be created:

- table or view
- object table or object view
- a column of nested-table type

If you omit *schema*, Oracle assumes the table is in your own schema. You can create triggers on index-organized tables.

Restriction: You cannot create a trigger on a table in the schema SYS.

NESTED TABLE — specifies that the trigger is being defined on column *nested_table_column* of a view. Such a trigger will fire only if the DML operates on the elements of the nested table.

Restriction: You can specify NESTED TABLE only for INSTEAD OF triggers.

DATABASE — specifies that the trigger is being defined on the entire database.

SCHEMA — specifies that the trigger is being defined on the current schema.

referencing_clause — specifies correlation names. You can use correlation names in the PL/SQL block and WHEN condition of a row trigger to refer specifically to old and new values of the current row. The default correlation names are OLD and NEW. If your row trigger is associated with a table named OLD or NEW, use this clause to specify different correlation names to avoid confusion between the table name and the correlation name.

- If the trigger is defined on a **nested table**, OLD and NEW refer to the row of the nested table, and PARENT refers to the current row of the parent table.
- If the trigger is defined on an object table or view, OLD and NEW refer to object instances.

Restriction: This clause is valid only for DML event triggers (not DDL or database event triggers).

FOR EACH ROW — designates the trigger to be a row trigger. Oracle fires a row trigger once for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the WHEN condition.

Note: This clause is applies only to DML events, not to DDL or database events.

Except for INSTEAD OF triggers, if you omit this clause, the trigger is a statement trigger. Oracle fires a statement trigger only once when the triggering statement is issued if the optional trigger constraint is met.

INSTEAD OF trigger statements are implicitly activated for each row.

WHEN (*condition*) specifies the trigger restriction, which is a SQL condition that must be satisfied for Oracle to fire the trigger. See the syntax description of condition in "Conditions" on page 5-13. This condition must contain correlation names and cannot contain a query.

Restrictions:

- You can specify a trigger restriction only for a row trigger. Oracle evaluates this condition for each row affected by the triggering statement.
- You cannot specify trigger restrictions for INSTEAD OF trigger statements.
- You can reference object columns or their attributes, or varray, nested table, or LOB columns. You cannot invoke PL/SQL functions or methods in the trigger restriction.

pl/sql_block is the PL/SQL block that Oracle executes to fire the trigger. For information on PL/SQL, including how to write PL/SQL blocks, see *PL/SQL User's Guide and Reference*.

The PL/SQL block of a database trigger can contain one of a series of built-in functions in the SYS schema designed solely to extract system event attributes. These functions can be used **only** in the PL/SQL block of a database trigger. For information on these functions, see *Oracle8i Application Developer's Guide - Fundamentals.*

Restrictions:

- The PL/SQL block of a trigger cannot contain transaction control SQL statements (COMMIT, ROLLBACK, SAVEPOINT, and SET CONSTRAINT) if the block is executed within the same transaction. For more information, see *Oracle8i Application Developer's Guide - Fundamentals.*
- You can reference and use LOB columns in the trigger action inside the PL/SQL block, but you cannot modify their values within the trigger action.

call_procedure_statement enables you to call a stored procedure, rather than specifying inline the trigger code as a PL/SQL block. The syntax of this statement is the same as that for "CALL" on page 7-210, with the following exceptions:

- You cannot specify the INTO clause of CALL, because it applies only to functions.
- You cannot specify bind variables in *expr*.
- To reference columns of tables on which the trigger is being defined, you must specify :NEW and :OLD. See the "Calling a Procedure in a Trigger Body Example" on page 7-408.

Examples

DML Trigger Example This example creates a BEFORE statement trigger named EMP_PERMIT_CHANGES in the schema SCOTT. You would write such a trigger

to place restrictions on DML statements issued on this table (such as when such statements could be issued).

```
CREATE TRIGGER scott.emp_permit_changes
    BEFORE
    DELETE OR INSERT OR UPDATE
    ON scott.emp
       pl/sql block
```

Oracle fires this trigger whenever a DELETE, INSERT, or UPDATE statement affects the EMP table in the schema SCOTT. The trigger EMP_PERMIT_CHANGES is a BEFORE statement trigger, so Oracle fires it once before executing the triggering statement.

DML Trigger Example with Restriction This example creates a BEFORE row trigger named SALARY_CHECK in the schema SCOTT. The PL/SQL block might specify, for example, that the employee's salary must fall within the established salary range for the employee's job:

```
CREATE TRIGGER scott.salary_check
    BEFORE
    INSERT OR UPDATE OF sal, job ON scott.emp
    FOR EACH ROW
    WHEN (new.job <> 'PRESIDENT')
       pl/sql_block
```

Oracle fires this trigger whenever one of the following statements is issued:

- an INSERT statement that adds rows to the EMP table
- an UPDATE statement that changes values of the SAL or JOB columns of the EMP table

SALARY_CHECK is a BEFORE row trigger, so Oracle fires it before changing each row that is updated by the UPDATE statement or before adding each row that is inserted by the INSERT statement.

SALARY_CHECK has a trigger restriction that prevents it from checking the salary of the company president.

Calling a Procedure in a Trigger Body Example You could create the SALARY_CHECK trigger described in the preceding example by calling a procedure instead of providing the trigger body in a PL/SQL block. Assume you have defined a procedure SCOTT.CHECK_SAL, which verifies that an employee's salary in in an appropriate range. Then you could create the trigger SALARY_CHECK as follows:

```
CREATE TRIGGER scott.salary_check
   BEFORE INSERT OR UPDATE OF sal, job ON scott.emp
   FOR EACH ROW
   WHEN (new.job<> 'PRESIDENT')
   CALL check_sal(:new.job, :new.sal, :new.ename);
```

The procedure CHECK_SAL could be implemented in PL/SQL, C, or Java. Also, you can specify :OLD values in the CALL clause instead of :NEW values.

Database Event Trigger Example This example creates a trigger to log all errors. The PL/SQL block does some special processing for a particular error (invalid logon, error number 1017. This trigger is an AFTER statement trigger, so it is fired after an unsuccessful statement execution (such as unsuccessful logon).

```
CREATE TRIGGER log_errors AFTER SERVERERROR ON DATABASE
   BEGIN
      IF (IS_SERVERERROR (1017)) THEN
         <special processing of logon error>
      ELSE
         <log error number>
      END IF;
   END;
```

DML Trigger Example This example creates an AFTER statement trigger on any DDL statement CREATE. Such a trigger can be used to audit the creation of new data dictionary objects in your schema.

```
CREATE TRIOGGER audit_db_object AFTER CREATE
   ON SCHEMA
      pl/sql_block
```

INSTEAD OF Trigger Example In this example, customer data is stored in two tables. The object view ALL_CUSTOMERS is created as a UNION of the two tables, CUSTOMERS_SJ and CUSTOMERS_PA. An INSTEAD OF trigger is used to insert values.

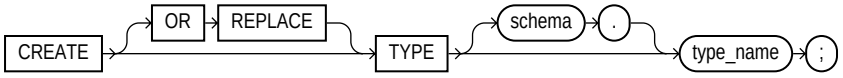
```
CREATE TABLE customers_sj
  ( cust    NUMBER(6),
    address VARCHAR2(50),
    credit   NUMBER(9,2)  );

CREATE TABLE customers_pa
  ( cust    NUMBER(6),
    address VARCHAR2(50),
    credit   NUMBER(9,2) );
```

```
CREATE TYPE customer_t AS OBJECT
  ( cust    NUMBER(6),
    address   VARCHAR2(50),
    credit    NUMBER(9,2),
    location   VARCHAR2(20)  );

CREATE VIEW all_customers (cust)
    AS SELECT customer_t (cust, address, credit, 'SAN_JOSE')
    FROM   customers_sj
  UNION ALL
    SELECT customer_t (cust, address, credit, 'PALO_ALTO')
    FROM   customers_pa;

CREATE TRIGGER instrig INSTEAD OF INSERT ON all_customers
   FOR EACH ROW
     BEGIN
      IF (:new.cust.location = 'SAN_JOSE') THEN
        INSERT INTO customers_sj
        VALUES (:new.cust.cust, :new.cust.address,:new.cust.credit);
      ELSE
       INSERT INTO customers_pa
       VALUES (:new.cust.cust, :new.cust.address, :new.cust.credit);
      END IF;
    END;
```

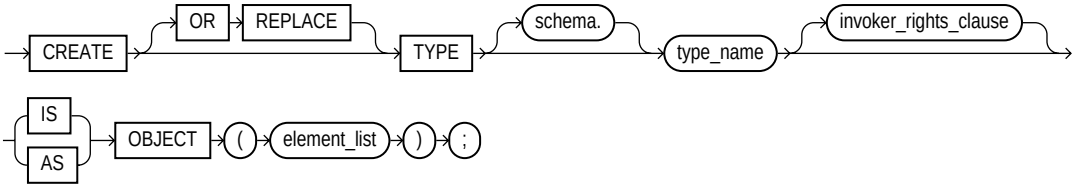
CREATE TYPE

Syntax

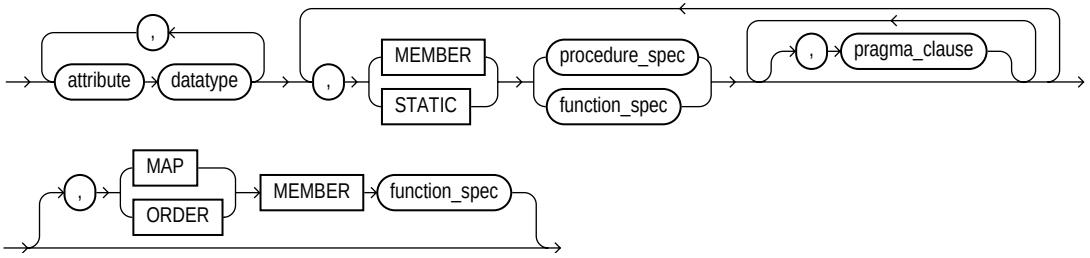
create_incomplete_type::=



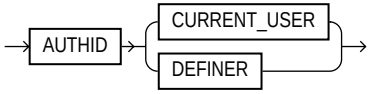
create_object_type::=



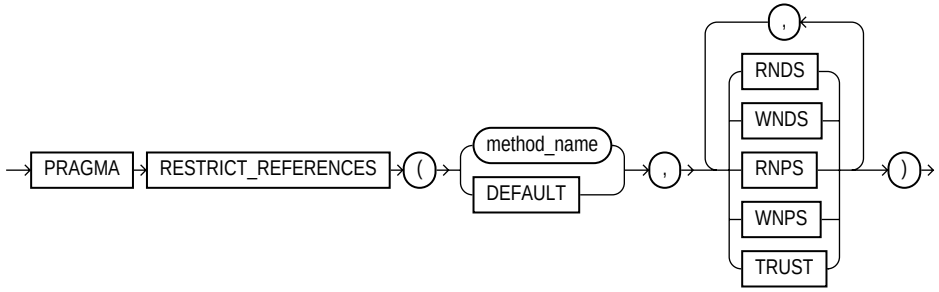
element_list::=



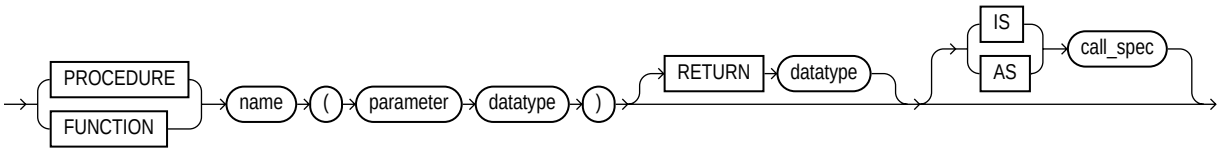
invoker_rights_clause::=



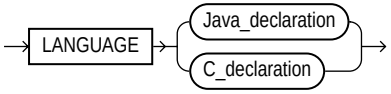
pragma_clause::=



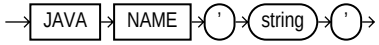
procedure_spec | function_spec::=



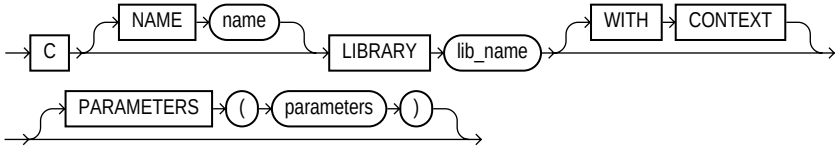
call_spec::=



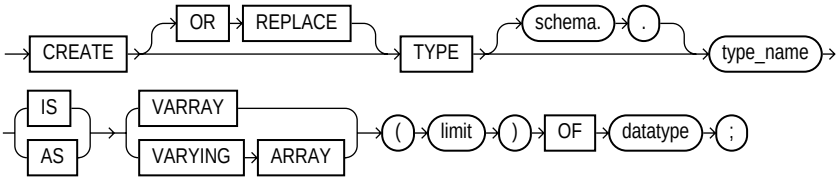
Java_declaration::=



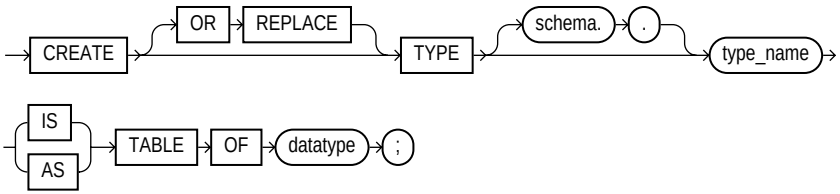
C_declaration::=



create_varray_type::=



create_nested_table_type::=



Purpose

To create an **object type**, named varying array (**varray**), **nested table type**, or an **incomplete object type**.

Oracle implicitly defines a constructor method for each user-defined type that you create. A **constructor** is a system-supplied procedure that is used in SQL statements or in PL/SQL code to construct an instance of the type value. The name of the constructor method is the same as the name of the user-defined type.

The parameters of the object type constructor method are the data attributes of the object type. They occur in the same order as the attribute definition order for the object type. The parameters of a nested table or varray constructor are the elements of the nested table or the varray.

An **incomplete type** is a type created by a forward type definition. It is called "incomplete" because it has a name but no attributes or methods. It can be referenced by other types, and so can be used to define types that refer to each other. However, you must fully specify the type before you can use it to create a table or an object column or a column of a nested table type.

For more information about objects, incomplete types, varrays, and nested tables see the *PL/SQL User's Guide and Reference*, *Oracle8i Application Developer's Guide - Fundamentals*, and *Oracle8i Concepts*.

Prerequisites

To create a type in your own schema, you must have the CREATE TYPE system privilege. To create a type in another user's schema, you must have the CREATE ANY TYPE system privilege. You can acquire these privileges explicitly or be granted them through a role.

The owner of the type must either be explicitly granted the EXECUTE object privilege in order to access all other types referenced within the definition of the type, or the type owner must be granted the EXECUTE ANY TYPE system privilege. The owner **cannot** obtain these privileges through roles.

If the type owner intends to grant other users access to the type, the owner must be granted the EXECUTE object privilege to the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. Otherwise, the type owner has insufficient privileges to grant access on the type to other users.

Keywords and Parameters

OR REPLACE	re-creates the type if it already exists. Use this clause to change the definition of an existing type without first dropping it. Users previously granted privileges on the re-created object type can use and reference the object type without being granted privileges again. If any function-based indexes depend on the type, Oracle marks the indexes DISABLED.
schema	is the schema to contain the type. If you omit *schema*, Oracle creates the type in your current schema.
type_name	is the name of an object type, a nested table type, or a varray type. If creating the type results in compilation errors, Oracle returns an error. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.
create_object_type	creates the type as a user-defined object type. The variables that form the data structure are called *attributes*. The member subprograms that define the object's behavior are called *methods*. AS OBJECT is required when creating an object type.

invoker_rights_clause specifies whether the member functions and procedures of the object type execute with the privileges and in the schema of the user who owns the object type or with the privileges and in the schema of CURRENT_USER. This specification applies to the corresponding type body as well. (For information on how CURRENT_USER is determined, see *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals*.)

This clause also determines how Oracle resolves external names in queries, DML operations, and dynamic SQL statements in the member functions and procedures of the type. For more information refer to *PL/SQL User's Guide and Reference*.

Restriction: You can specify this clause only for an object type, not for a nested table or varray type.

AUTHID CURRENT_USER	specifies that the member functions and procedures of the object type execute with the privileges of CURRENT_USER. This clause creates an "invoker-rights type." This clause also specifies that external names in queries, DML operations, and dynamic SQL statements resolve in the schema of CURRENT_USER. External names in all other statements resolve in the schema in which the type resides.
AUTHID DEFINER	specifies that the member functions and procedures of the object type execute with the privileges of the owner of the schema in which the functions and procedures reside, and that external names resolve in the schema where the member functions and procedures reside. This is the default.

datatype is the name of the attribute's Oracle built-in datatype or user-defined type. For a list of possible datatypes, see "Datatypes" on page 2-5.

Restrictions:

- You cannot specify attributes of type ROWID, LONG, or LONG ROW.
- You cannot create an object with NCLOB, NCHAR, or NVARCHAR2 attributes, but you can specify parameters of these datatypes in methods.
- You cannot specify a datatype of UROWID for a user-defined object type.
- If you specify an object of type REF, the target object must have an object identifier.

attribute specifies, for an object type, the name of an object attribute. Attributes are data items with a name and a type specifier that form the structure of the object. You must specify at least one attribute for each object type.

MEMBER specifies a function or procedure subprogram associated with the object type that is referenced as an attribute. Typically you invoke member methods in a "selfish" style, such as `object_expression.method()`. This class of method has an implicit first argument referenced as SELF in the method's body, which represents the object on which the method has been invoked.

STATIC — also specifies a function or procedure subprogram associated with the object type. However, unlike member methods, static methods do not have any implicit parameters (that is, SELF is not referenceable in their body). They are typically invoked as `type_name.method()`.

For both member and static methods, you must specify a corresponding method body in the object type body for each procedure or function specification. See "CREATE TYPE BODY" on page 7-421. For information about method invocation and methods, see *PL/SQL User's Guide and Reference*.

procedure_spec | function_spec — is the specification of a procedure or function subprogram. The RETURN clause is valid only for a function. The syntax shown is an abbreviated form. For the full syntax with all possible clauses, see "CREATE PROCEDURE" on page 7-333 and "CREATE FUNCTION" on page 7-266.

If this subprogram does not include the declaration of the procedure or function, you must issue a corresponding CREATE TYPE BODY statement. See "CREATE TYPE BODY" on page 7-421.

For a list of restrictions on user-defined functions, see "Restrictions on User-Defined Functions" on page 7-268.

call_spec — is the call specification ("call spec") that maps a Java or C method name, parameter types, and return type to their SQL counterparts. If all the member methods in the type have been defined in this clause, you need not issue a corresponding CREATE TYPE BODY statement.

- In *Java_declaration*, *'string'* identifies the Java implementation of the method. For more information, see *Oracle8i Java Stored Procedures Developer's Guide*.
- For an explanation of the parameters and semantics of *the C_declaration*, see *Oracle8i Application Developer's Guide - Fundamentals*.

pragma_clause — specifies a compiler directive.

PRAGMA RESTRICT_REFERENCES — is a compiler directive that denies member functions read / write access to database tables, packaged variables, or both, and thereby helps to avoid side effects. For more information, see *Oracle8i Application Developer's Guide - Fundamentals*.

method_name — is the name of the MEMBER function or procedure to which the pragma is being applied.

DEFAULT — specifies that the pragma should be applied to all methods in the type for which a pragma has not been explicitly specified.

WNDS — specifies the constraint *writes no database state* (does not modify database tables).

WNPS — specifies the constraint *writes no package state* (does not modify packaged variables).

RNDS — specifies the constraint *reads no database state* (does not query database tables).

RNPS — specifies the constraint *reads no package state* (does not reference packages variables).

	TRUST	specifies that the restrictions listed in the pragma are not actually to be enforced, but are simply trusted to be true.

MAP MEMBER *function_spec*

specifies a member function (map method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined *scalar* type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the map method is null, the map method returns null and the method is not invoked.

An object specification can contain only one map method, which must be a function. The result type must be a predefined SQL scalar type, and the map function can have no arguments other than the implicit SELF argument.

Note: If *type_name* will be referenced in queries involving sorts (through an ORDER BY, GROUP BY, DISTINCT, or UNION clause) or joins, and you want those queries to be parallelized, you must specify a MAP member function.

ORDER MEMBER *function_spec*

specifies a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

If either argument to the order method is null, the order method returns null and the method is not invoked.

When instances of the same object type definition are compared in an ORDER BY clause, the order method *function_specification* is invoked.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

You can define either a MAP method or an ORDER method in a type specification, but not both. If you declare either method, you can compare object instances in SQL.

If neither a MAP nor an ORDER method is specified, only comparisons for equality or inequality can be performed. Therefore object instances cannot be ordered. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method needs to be specified to determine the equality of two object types.

Use MAP if you are performing extensive sorting or hash join operations on object instances. MAP is applied once to map the objects to scalar values and then the scalars are used during sorting and merging. A MAP method is more efficient than an ORDER method, which must invoke the method for each object comparison. You must use a MAP method for hash joins. You cannot use an ORDER method because the hash mechanism hashes on the object value. For more information about object value comparisons, see *Oracle8i Application Developer's Guide - Fundamentals*.

create_varray_type

creates the type as an ordered set of elements, each of which has the same datatype. You must specify a name and a maximum *limit* of zero or more. The array limit must be an integer literal. Oracle does not support anonymous varrays.

The type name for the objects contained in the varray must be one of the following:

- A built-in datatype,
- A REF, or
- An object type, including an object with varray attributes.

The type name for the objects contained in the varray cannot be

- An object type with a nested table attribute,
- A varray type, or
- A TABLE type.

Restrictions:

- A collection type cannot contain any other collection type, either directly or indirectly. That is, a varray type cannot contain any elements that are varrays or nested tables.
- You cannot create varray types of LOB datatypes.

create_nested_table_type creates a named nested table of type *datatype*.

When *datatype* is an object type, the nested table type describes a table whose columns match the name and attributes of the object type.

When *datatype* is a scalar type, then the nested table type describes a table with a single, scalar type column called "column_value".

Restrictions:

- A collection type cannot contain any other collection type, either directly or indirectly. That is, a nested table type cannot contain any elements that are varrays or nested tables.
- You cannot specify NCLOB for *datatype*. However, you can specify CLOB or BLOB.

Examples

Object Type Example The following example creates object type PERSON_T with LOB attributes:

```
CREATE TYPE person_t AS OBJECT
  (name CHAR(20),
   resume CLOB,
   picture BLOB);
```

Varray Type Example The following statement creates MEMBERS_TYPE as a varray type with 100 elements:

```
CREATE TYPE members_type AS VARRAY(100) OF CHAR(5);
```

Nested Table Type Example The following example creates a named table type PROJECT_TABLE of object type PROJECT_T:

```
CREATE TYPE project_t AS OBJECT
  (pno CHAR(5),
   pname CHAR(20),
   budgets DEC(7,2));

CREATE TYPE project_table AS TABLE OF project_t;
```

Constructor Example The following example invokes method constructor COL.GETBAR():

```
CREATE TYPE foo AS OBJECT (a1 NUMBER,
                  MEMBER FUNCTION getbar RETURN NUMBER,);
CREATE TABLE footab(col foo);

SELECT col.getbar() FROM footab;
```

Unlike function invocations, method invocations require parentheses, even when the methods do not have additional arguments.

The next example invokes the system-defined constructor to construct the FOO_T object and insert it into the FOO_TAB table:

```
CREATE TYPE foo_t AS OBJECT (a1 NUMBER, a2 NUMBER);
CREATE TABLE foo_tab (b1 NUMBER, b2 foo_t);
INSERT INTO foo_tab VALUES (1, foo_t(2,3));
```

For more information about constructors, see *Oracle8i Application Developer's Guide - Fundamentals* and *PL/SQL User's Guide and Reference.*

Static Method Example The following example changes the definition of the EMPLOYEE_T type to associate it with the CONSTRUCT_EMP function:

```
CREATE OR REPLACE TYPE employee_t AS OBJECT(
   empid RAW(16),
   ename CHAR(31),
   dept   REF department_t,
   STATIC function construct_emp
      (name VARCHAR2, dept REF department_t)
   RETURN employee_t
  );
```

This statement requires the following type body statement:

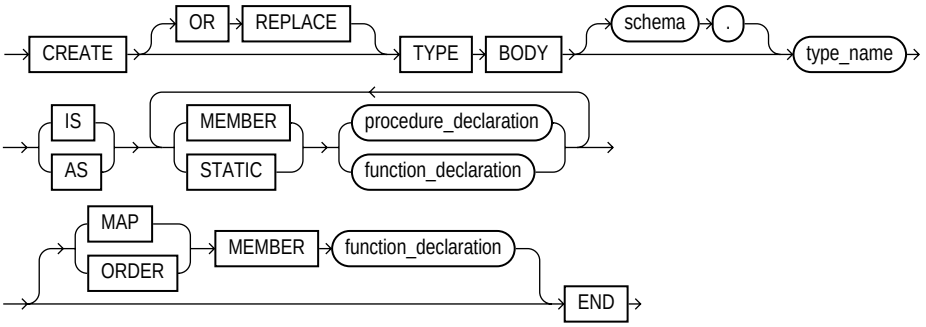
```
CREATE OR REPLACE TYPE BODY employee_t IS
   STATIC FUNCTION construct_emp
      (name varchar2, dept REF department_t)
   RETURN employee_t IS
      BEGIN
         return employee_t(SYS_GUID(),name,dept);
      END;
   END;
```

This type and type body definition allows the following operation:

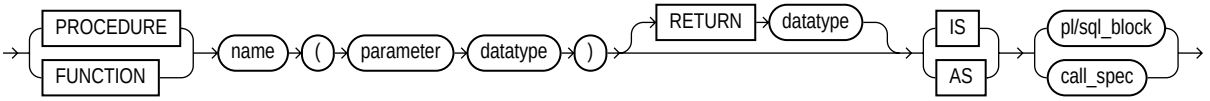
```
INSERT INTO emptab
   VALUES (employee_t.construct_emp('John Smith', NULL));
```

CREATE TYPE BODY

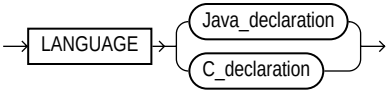
Syntax



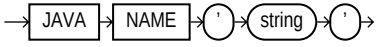
procedure_declaration | function_declaration::=



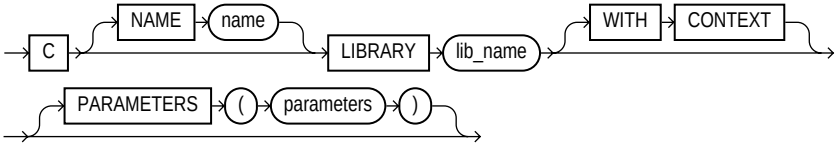
call_spec::=



Java_declaration::=



C_declaration::=



Purpose

To define or implement the member methods defined in the object type specification. You create object types with the CREATE TYPE and the CREATE TYPE BODY statements. The CREATE TYPE statement specifies the name of the object type, its attributes, methods, and other properties. The CREATE TYPE BODY statement contains the code for the methods in the type.

For each method specified in an object type specification for which you did not specify the *call_spec*, you must specify a corresponding method body in the object type body.

For information on creating and modifying a type specification, see "CREATE TYPE" on page 7-411 and "ALTER TYPE" on page 7-173.

Prerequisites

Every member declaration in the CREATE TYPE specification for object types must have a corresponding construct in the CREATE TYPE or CREATE TYPE BODY statement.

To create or replace a type body in your own schema, you must have the CREATE TYPE or the CREATE ANY TYPE system privilege. To create an object type in another user's schema, you must have the CREATE ANY TYPE system privileges. To replace an object type in another user's schema, you must have the DROP ANY TYPE system privileges.

Keywords and Parameters

OR REPLACE	re-creates the type body if it already exists. Use this clause to change the definition of an existing type body without first dropping it. Users previously granted privileges on the re-created object type body can use and reference the object type body without being granted privileges again. You can use this clause to add new member subprogram definitions to specifications added with the ALTER TYPE ... REPLACE statement.
schema	is the schema to contain the type body. If you omit *schema*, Oracle creates the type body in your current schema.
type_name	is the name of an object type.
MEMBER | STATIC	declares or implements a method function or procedure subprogram associated with the object type specification. For a description of the difference between member and static methods, see "CREATE TYPE" on page 7-411. For information about overloading subprogram names within a package, see *PL/SQL User's Guide and Reference*.

You must define a corresponding method name, optional parameter list, and (for functions) a return type in the object type specification for each procedure or function declaration.

procedure_declaration is the declaration of a procedure subprogram.

function_declaration is the declaration of a function subprogram.

For more information, see "CREATE PROCEDURE" on page 7-333, "CREATE FUNCTION" on page 7-266, and *Oracle8i Application Developer's Guide - Fundamentals*.

MAP MEMBER declares or implements a member function (MAP method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and specifies an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the map method is null, the map method returns null and the method is not invoked.

An object type body can contain only one map method, which must be a function. The map function can have no arguments other than the implicit SELF argument.

ORDER MEMBER specifies a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

If either argument to the order method is null, the order method returns null and the method is not invoked.

When instances of the same object type definition are compared in an ORDER BY clause, Oracle invokes the order method *function_spec*.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

You can declare either a MAP method or an ORDER method, but not both. If you declare either method, you can compare object instances in SQL.

If you do not declare either method, you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal.

procedure_declaration | *function_declaration* is the declaration of a procedure or function subprogram. The RETURN clause is valid only for a function. The syntax shown is an abbreviated form. For the full syntax with all possible clauses, see "CREATE PROCEDURE" on page 7-333 and "CREATE FUNCTION" on page 7-266.

pl/sql_block declares the procedure or function. For more information, see *PL/SQL User's Guide and Reference*.

call_spec	is the call specification ("call spec") that maps a Java or C method name, parameter types, and return type to their SQL counterparts. ■ In *Java_declaration*, *'string*' identifies the Java implementation of the method. For more information, see *Oracle8i Java Stored Procedures Developer's Guide*. ■ For an explanation of the parameters and semantics of *the C_ declaration*, see *Oracle8i Application Developer's Guide - Fundamentals*.
AS EXTERNAL	is an alternative way of declaring a C method. This clause has been deprecated and is supported for backward compatibility only. Oracle Corporation recommends that you use the AS LANGUAGE C syntax.

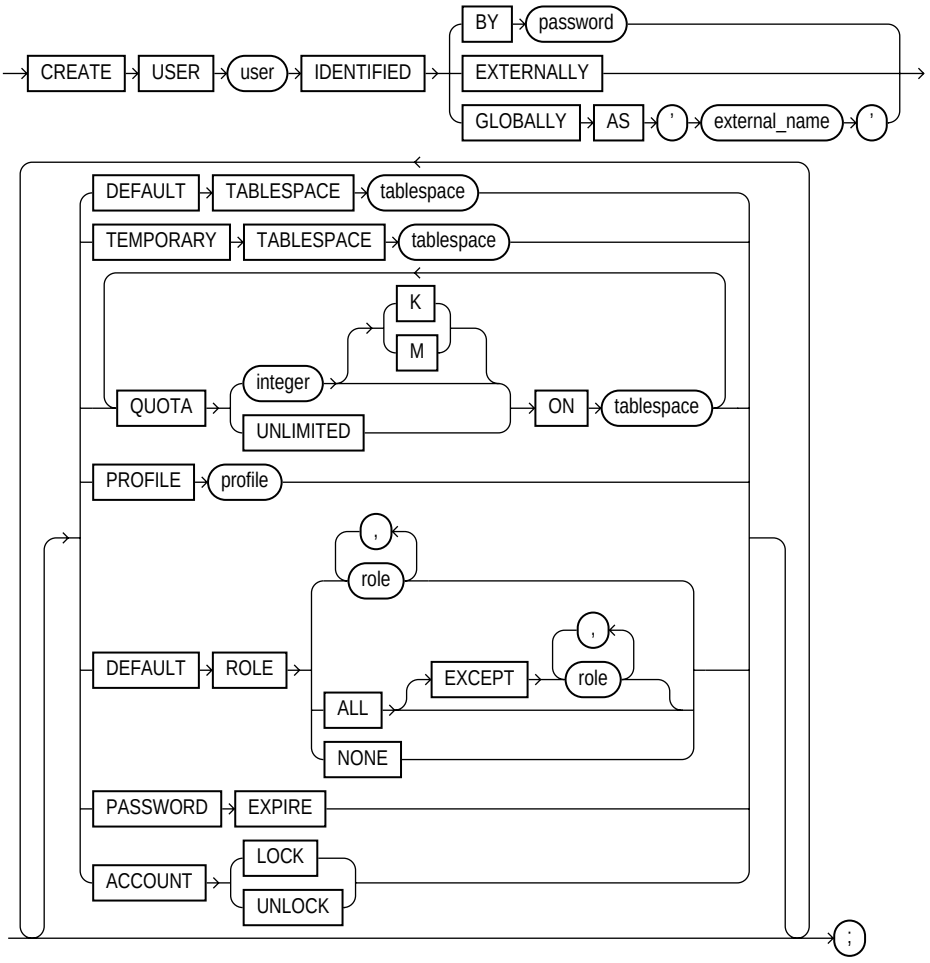
Examples

The following object type body implements member subprograms for RATIONAL.

```
CREATE TYPE BODY rational
  IS
   MAP MEMBER FUNCTION rat_to_real RETURN REAL IS
      BEGIN
         RETURN numerator/denominator;
      END;

   MEMBER PROCEDURE normalize IS
      gcd NUMBER := integer_operations.greatest_common_divisor
                    (numerator, denominator);
      BEGIN
         numerator := numerator/gcd;
         denominator := denominator/gcd;
      END;

   MEMBER FUNCTION plus(x rational) RETURN rational IS
      r rational := rational_operations.make_rational
                     (numerator*x.denominator +
                      x.numerator*denominator,
                      denominator*x.denominator);
      BEGIN
         RETURN r;
      END;

   END;
```

CREATE USER

Syntax



Purpose

To create and configure a database **user**, or an account through which you can log in to the database and establish the means by which Oracle permits access by the user.

> **Note:** You can enable a user to connect to Oracle through an proxy (that is, an application or application server). For syntax and discussion, refer to "ALTER USER" on page 7-179.

Prerequisites

You must have CREATE USER system privilege.

When you create a user with the CREATE USER statement, the user's privilege domain is empty. To log on to Oracle, a user must have CREATE SESSION system privilege. Therefore, after creating a user, you should grant the user at least the CREATE SESSION privilege. See "GRANT system_privileges_and_roles" on page 7-493.

Keywords and Parameters

user is the name of the user to be created. This name can contain only characters from your database character set and must follow the rules described in the section "Schema Object Naming Rules" on page 2-67. Oracle recommends that the user name contain at least one single-byte character regardless of whether the database character set also contains multi-byte characters.

IDENTIFIED indicates how Oracle authenticates the user. See *Oracle8i Application Developer's Guide - Fundamentals* and your operating system specific documentation for more information.

BY *password* creates a **local user** and indicates that the user must specify *password* to log on. Passwords can contain only single-byte characters from your database character set regardless of whether this character set also contains multibyte characters.

Passwords must follow the rules described in the section "Schema Object Naming Rules" on page 2-67, unless you are using Oracle's password complexity verification routine. That routine requires a more complex combination of characters than the normal naming rules permit. You implement this routine with the UTLPWDMG.SQL script, which is further described in *Oracle8i Administrator's Guide*.

Also refer to *Oracle8i Administrator's Guide* for a detailed description and explanation of how to use password management and protection.

	EXTERNALLY	creates an **external user** and indicates that a user must be authenticated by an external service (such as an operating system or a third-party service). Doing so causes Oracle to rely on the login authentication of the operating system to ensure that a specific operating system user has access to a specific database user. **WARNING: Oracle strongly recommends that you do not use IDENTIFIED EXTERNALLY with operating systems that have inherently weak login security. For more information, see** *Oracle8i Administrator's Guide***.**
	GLOBALLY AS '*external_name*'	creates a **global user** and indicates that a user must be authenticated by the enterprise directory service. The '*external_name*' string is the X.509 name at the enterprise directory service that identifies this user. It should be of the form 'CN=*username,other_attributes*', where *other_attributes* is the rest of the user's distinguished name (DN) in the directory.
	Note: You can control the ability of an application server to connect as the specified user and to activate that user's roles using the ALTER USER statement. See "ALTER USER" on page 7-179	
DEFAULT TABLESPACE	identifies the default tablespace for objects that the user creates. If you omit this clause, objects default to the SYSTEM tablespace. For more information on tablespaces, see "CREATE TABLESPACE" on page 7-394.	
TEMPORARY TABLESPACE	identifies the tablespace for the user's temporary segments. If you omit this clause, temporary segments default to the SYSTEM tablespace.	
QUOTA	allows the user to allocate space in the tablespace and optionally establishes a quota of *integer* bytes. Use K or M to specify the quota in kilobytes or megabytes. This quota is the maximum space in the tablespace the user can allocate. A CREATE USER statement can have multiple QUOTA clauses for multiple tablespaces.	
	UNLIMITED	allows the user to allocate space in the tablespace without bound.
PROFILE	reassigns the profile named to the user. The profile limits the amount of database resources the user can use. If you omit this clause, Oracle assigns the DEFAULT profile to the user. See also "GRANT system_privileges_and_roles" on page 7-493 and "CREATE PROFILE" on page 7-338.	
DEFAULT ROLE	lets you assign and enable a default role or roles to the user. ■ *role* assigns one or more predefined roles ■ ALL [EXCEPT] *role* assigns all predefined roles to the user, or all except those specified. ■ NONE assigns no roles to the user.	
PASSWORD EXPIRE	causes the user's *password* to expire. This setting forces the user (or the DBA) to change the password before the user can log in to the database.	

ACCOUNT LOCK	locks the user's account and disables access.
ACCOUNT UNLOCK	unlocks the user's account and enables access to the account.

Examples

If you create a new user with PASSWORD EXPIRE, the user's password must be changed before attempting to log in to the database. You can create the user SIDNEY by issuing the following statement:

```
CREATE USER sidney
    IDENTIFIED BY welcome
    DEFAULT TABLESPACE cases_ts
    QUOTA 10M ON cases_ts
    QUOTA 5M ON temp_ts
    QUOTA 5M ON system
    PROFILE engineer
    PASSWORD EXPIRE;
```

The user SIDNEY has the following characteristics:

- The password WELCOME
- Default tablespace CASES_TS, with a quota of 10 megabytes
- Temporary tablespace TEMP_TS, with a quota of 5 megabytes
- Access to the tablespace SYSTEM, with a quota of 5 megabytes
- Limits on database resources defined by the profile ENGINEER
- An expired password, which must be changed before SIDNEY can log in to the database

To create a user accessible only by the operating system account GEORGE, prefix GEORGE by the value of the initialization parameter OS_AUTHENT_PREFIX. For example, if this value is "OPS$", you can create the user OPS$GEORGE with the following statement:

```
CREATE USER ops$george
    IDENTIFIED EXTERNALLY
    DEFAULT TABLESPACE accs_ts
    TEMPORARY TABLESPACE temp_ts
    QUOTA UNLIMITED ON accs_ts
    QUOTA UNLIMITED ON temp_ts;
```

The user OPS$GEORGE has the following additional characteristics:

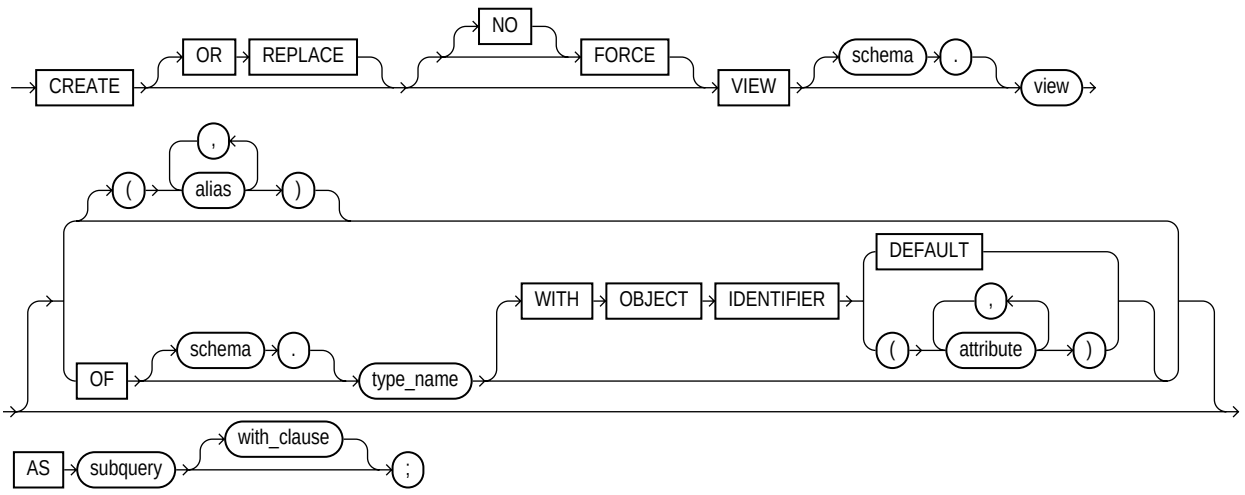
- Default tablespace ACCS_TS
- Default temporary tablespace TEMP_TS
- Unlimited space on the tablespaces ACCS_TS and TEMP_TS
- Limits on database resources defined by the DEFAULT profile

The following example creates user CINDY as a global user:

```
CREATE USER cindy
   IDENTIFIED GLOBALLY AS 'CN=cindy,OU=division1,O=oracle,C=US'
   DEFAULT TABLESPACE legal_ts
   QUOTA 20M ON legal_ts
   PROFILE lawyer;
```

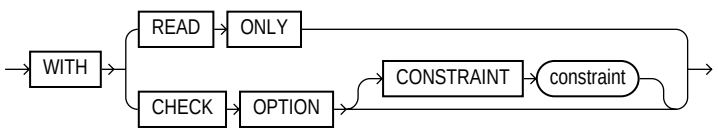
CREATE VIEW

Syntax



subquery: See "SELECT and Subqueries" on page 7-541.

with_clause::=



Purpose

To define a **view**, a logical table based on one or more tables or views. A view contains no data itself. The tables upon which a view is based are called **base tables**.

You can also create an **object view** or a relational view that supports LOB and object datatypes (object types, REFs, nested table, or varray types) on top of the existing view mechanism. An object view is a view of a user-defined type, where each row contains objects, each object with a unique object identifier.

For information on various types of views and their uses, see *Oracle8i Concepts*, *Oracle8i Application Developer's Guide - Fundamentals*, and *Oracle8i Administrator's Guide.*

For information on modifying a view, see "ALTER VIEW" on page 7-183. For information on removing a view from the database, see "DROP VIEW" on page 7-485.

Prerequisites

To create a view in your own schema, you must have CREATE VIEW system privilege. To create a view in another user's schema, you must have CREATE ANY VIEW system privilege.

The owner of the schema containing the view must have the privileges necessary to either select, insert, update, or delete rows from all the tables or views on which the view is based. For information on these privileges, see "SELECT and Subqueries" on page 7-541, "INSERT" on page 7-512, "UPDATE" on page 7-584, and "DELETE" on page 7-438. The owner must be granted these privileges directly, rather than through a role.

To use the basic constructor method of an object type when creating an object view, one of the following must be true:

- The object type must belong to the same schema as the view to be created.
- You must have EXECUTE ANY TYPE system privileges.
- You must have the EXECUTE object privilege on that object type.

Partition Views

Partition views were introduced in Release 7.3 to provide partitioning capabilities for applications requiring them. Partition views are supported in Oracle8*i* so that you can upgrade applications from Release 7.3 without any modification. In most cases, subsequent to migration to Oracle8*i* you will want to migrate partition views into partitions (see *Oracle8i Administrator's Guide*).

With Oracle8*i*, you can use the CREATE TABLE statement to create partitioned tables easily. Partitioned tables offer the same advantages as partition views, while also addressing their shortcomings. For more information on the shortcomings of partition reviews, see *Oracle8i Concepts.*

Oracle recommends that you use partitioned tables rather than partition views in most operational environments. For more information about partitioned tables, see "CREATE TABLE" on page 7-359.

Keywords and Parameters

OR REPLACE	re-creates the view if it already exists. You can use this clause to change the definition of an existing view without dropping, re-creating, and regranting object privileges previously granted on it. INSTEAD OF triggers defined in the view are dropped when a view is re-created. See "CREATE TRIGGER" on page 7-401 for more information about the INSTEAD OF clause. **Note:** If any materialized views are dependent on *view*, those materialized views will be marked INVALID and UNUSABLE and will require a full refresh to restore them to a usable state. Invalid materialized views cannot be used by query rewrite and cannot be refreshed until they are recompiled. For information on refreshing invalid materialized views, see "ALTER MATERIALIZED VIEW / SNAPSHOT" on page 7-45. For information on materialized views in general, see *Oracle8i Concepts*.
FORCE	creates the view regardless of whether the view's base tables or the referenced object types exist or the owner of the schema containing the view has privileges on them. These conditions must be true before any SELECT, INSERT, UPDATE, or DELETE statements can be issued against the view.
NO FORCE	creates the view only if the base tables exist and the owner of the schema containing the view has privileges on them. This is the default.
schema	is the schema to contain the view. If you omit *schema*, Oracle creates the view in your own schema.
view	is the name of the view or the object view. **Restriction:** If a view has INSTEAD OF triggers, any views created on it must have INSTEAD OF triggers, even if the views are inherently updatable.
alias	specifies names for the expressions selected by the view's query. The number of aliases must match the number of expressions selected by the view. Aliases must follow the rules for naming schema objects in the section, "Referring to Schema Objects and Parts" on page 2-71. Aliases must be unique within the view. If you omit the aliases, Oracle derives them from the columns or column aliases in the view's query. For this reason, you **must** use aliases if the view's query contains expressions rather than only column names. **Restriction:** You cannot specify an alias when creating an object view.
OF *type_name*	explicitly creates an object view of type *type_name*. The columns of an object view correspond to the top-level attributes of type *type_name*. Each row will contain an object instance and each instance will be associated with an object identifier (OID) as specified in the WITH OBJECT IDENTIFIER clause. If you omit *schema*, *Oracle* creates the object view in your own schema. For more information about creating objects, see "CREATE TYPE" on page 7-411.

WITH OBJECT IDENTIFIER — specifies the attributes of the object type that will be used as a key to identify each row in the object view. In most cases these attributes correspond to the primary-key columns of the base table. You must ensure that the attribute list is unique and identifies exactly one row in the view.

If you try to dereference or pin a primary key REF that resolves to more than one instance in the object view, Oracle raises an error.

Note: The 8.0 syntax WITH OBJECT OID is replaced with this syntax for clarity. The keywords WITH OBJECT OID are supported for backward compatibility, but Oracle Corporation recommends that you use the new syntax WITH OBJECT IDENTIFIER.

If the object view is defined on an object table or an object view, you can omit this clause or specify DEFAULT.

DEFAULT — specifies that the intrinsic object identifier of the underlying object table or object view will be used to uniquely identify each row.

attribute — is an attribute of the object type from which the object identifier for the object view is to be created.

AS *subquery* — identifies columns and rows of the table(s) that the view is based on. The subquery's select list can contain up to 1000 expressions.

If you create views that refer to remote tables and views, the database links you specify must have been created using the CONNECT TO clause of the CREATE DATABASE LINK statement, and you must qualify them with schema name in the view query.

Restrictions:

- The view query cannot select the CURRVAL or NEXTVAL pseudocolumns.
- If the view query selects the ROWID, ROWNUM, or LEVEL pseudocolumns, those columns must have aliases in the view query.
- If the view query uses an asterisk (*) to select all columns of a table, and you later add new columns to the table, the view will not contain those columns until you re-create the view by issuing a CREATE OR REPLACE VIEW statement.
- For object views, the number of elements in the view subquery select list must be the same as the number of top-level attributes for the object type. The datatype of each of the selecting elements must be the same as the corresponding top-level attribute.

The preceding restrictions apply to materialized views as well.

- If you want the view to be inherently updatable, it must not contain any of the following constructs:

  - A set operator

  - A DISTINCT operator

  - An aggregate function

  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause

  - A collection expression in a SELECT list

  - A subquery in a SELECT list

  - Joins (with some exceptions). See *Oracle8i Administrator's Guide* for details.

- If an inherently updatable view contains pseudocolumns or expressions, the UPDATE statement must not refer to any of these pseudocolumns or expressions.

- If you want a join view to be updatable, all of the following conditions must be true:

  - The DML statement must affect only one table underlying the join.

  - For an UPDATE statement, all columns updated must be extracted from a key-preserved table. If the view has the CHECK OPTION, join columns and columns taken from tables that are referenced more than once in the view must be shielded from UPDATE.

  - For a DELETE statement, the join can have one and only one key-preserved table. That table can appear more than once in the join, unless the view has the CHECK OPTION.

  - For an INSERT statement, all columns into which values are inserted must come from a key-preserved table, and the view must not have the CHECK OPTION.

For more information on updatable views, see *Oracle8i Administrator's Guide*. For more information about updating object views or relational views that support object types, see *Oracle8i Application Developer's Guide - Fundamentals*.

with_clause restricts the subquery in one of the following ways:

WITH READ ONLY — specifies that no delete, inserts, or updates can be performed through the view.

WITH CHECK OPTION — specifies that inserts and updates performed through the view must result in rows that the view query can select. The CHECK OPTION cannot make this guarantee if:

- There is a subquery in the query of this view or any view on which this view is based or
- INSTEAD OF triggers.

CONSTRAINT *constraint*	assigns the name of the CHECK OPTION constraint. If you omit this identifier, Oracle automatically assigns the constraint a name of the form SYS_C*n*, where *n* is an integer that makes the constraint name unique within the database.

Examples

Basic View Example The following statement creates a view of the EMP table named DEPT20. The view shows the employees in Department 20 and their annual salary:

```
CREATE VIEW dept20
    AS SELECT ename, sal*12 annual_salary
        FROM emp
        WHERE deptno = 20;
```

The view declaration need not define a name for the column based on the expression SAL*12, because the subquery uses a column alias (ANNUAL_SALARY) for this expression.

Updatable View Example The following statement creates an updatable view named CLERKS of all clerks in the EMP table. Only the employees' IDs, names, and department numbers are visible in this view and only these columns can be updated in rows identified as clerks:

```
CREATE VIEW clerk (id_number, person, department, position)
    AS SELECT empno, ename, deptno, job
       FROM emp
       WHERE job = 'CLERK'
    WITH CHECK OPTION CONSTRAINT wco;
```

Because of the CHECK OPTION, you cannot subsequently insert a new row into CLERK if the new employee is not a clerk.

Join View Example A join view is one whose view subquery contains a join. If at least one column in the subquery join has a unique index, then it may be possible to modify one base table in a join view. You can query USER_UPDATABLE_COLUMNS to see whether the columns in a join view are updatable. For example:

```
CREATE VIEW ed AS
   SELECT e.empno, e.ename, d.deptno, d.loc
      FROM emp e, dept d
   WHERE e.deptno = d.deptno
```

```
View created.

SELECT column_name, updatable
  FROM user_updatable_columns
  WHERE table_name = 'ED';

COLUMN_NAME     UPD
--------------- ---
ENAME           YES
DEPTNO          NO
EMPNO           YES
LOC             NO

INSERT INTO ed (ENAME, EMPNO) values ('BROWN', 1234);
```

In the above example, there is a unique index on the DEPTNO column of the DEPT table. You can insert, update or delete a row from the EMP base table, because all the columns in the view mapping to the EMP table are marked as updatable and because the primary key of EMP is included in the view. For more information on updating join views, see the *Oracle8i Application Developer's Guide - Fundamentals*.

Note: You cannot insert into the table using the view unless the view contains all NOT NULL columns of all tables in the join, unless you have specified DEFAULT values for the NOT NULL columns.

Read-Only View Example The following statement creates a read-only view named CLERKS of all clerks in the EMP table. Only the employee's IDs, names, department numbers, and jobs are visible in this view:

```
CREATE VIEW clerk (id_number, person, department, position)
     AS SELECT empno, ename, deptno, job
       FROM emp
       WHERE job = 'CLERK'
    WITH READ ONLY;
```

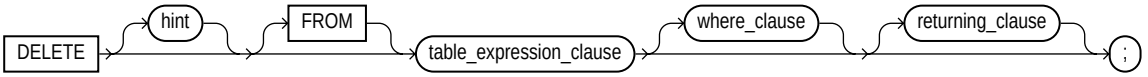
Object View Example The following example creates object view EMP_OBJECT_VIEW of EMPLOYEE_TYPE:

```
CREATE TYPE employee_type AS OBJECT
  ( empno       NUMBER(4),
    ename       VARCHAR2(20),
```

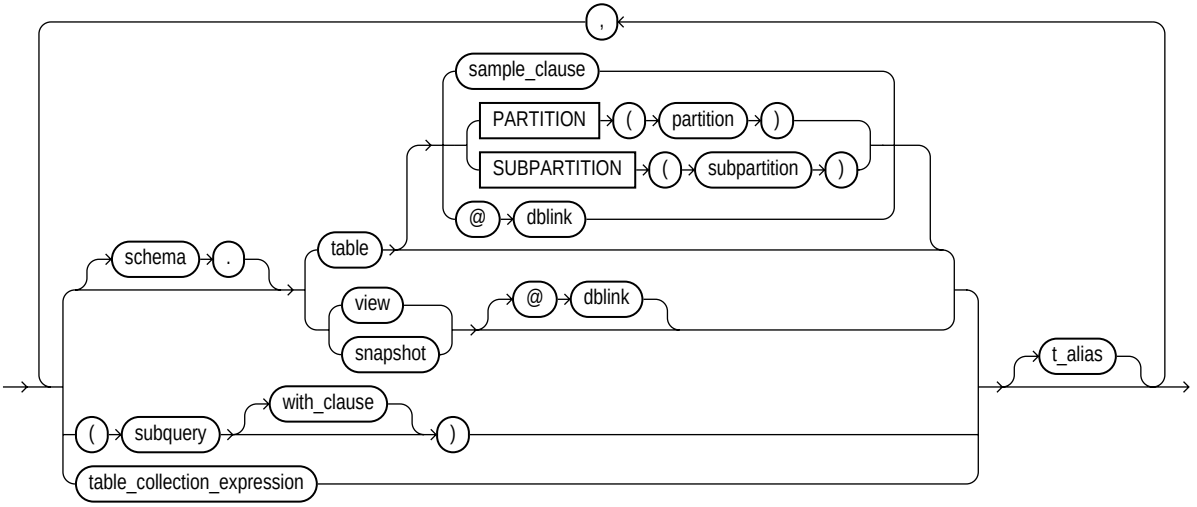
```
    job         VARCHAR2(9),
    mgr         NUMBER(4),
    hiredate    DATE,
    sal         NUMBER(7,2),
    comm        NUMBER(7,2)  );

CREATE OR REPLACE VIEW emp_object_view OF employee_type
  WITH OBJECT IDENTIFIER (empno)
  AS SELECT empno, ename, job, mgr, hiredate, sal, comm
     FROM emp;
```

DELETE

Syntax

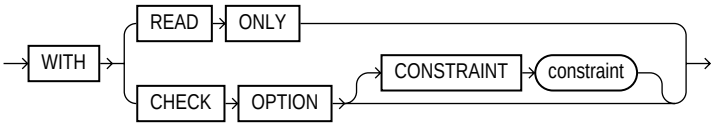


table_expression_clause::=



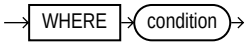
subquery: See "SELECT and Subqueries" on page 7-541.

with_clause::=

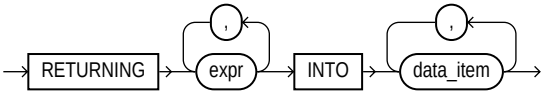


table_collection_expression::=

where_clause::=



returning_clause::=



Purpose

To remove rows from a table, a partitioned table, a view's base table, or a view's partitioned base table.

Prerequisites

For you to delete rows from a table, the table must be in your own schema or you must have DELETE privilege on the table.

For you to delete rows from the base table of a view, the owner of the schema containing the view must have DELETE privilege on the base table. Also, if the view is in a schema other than your own, you must be granted DELETE privilege on the view.

The DELETE ANY TABLE system privilege also allows you to delete rows from any table or table partition, or any view's base table.

If the SQL92_SECURITY initialization parameter is set to TRUE, then you must have SELECT privilege on the table to perform a DELETE that references table columns (such as the columns in a *where_clause*).

Keywords and Parameters

hint is a comment that passes instructions to the optimizer on choosing an execution plan for the statement. For the syntax and description of hints, see "Hints" on page 2-58 and *Oracle8i Tuning*.

table_expression_clause

schema is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table | view | subquery is the name of a table or view, or the column or columns resulting from a subquery, from which the rows are to be deleted. If you specify *view*, Oracle deletes rows from the view's base table.

If table (or the base table of view) contains one or more domain index columns, this statements executes the appropriate indextype delete routine. For more information on these routines, see *Oracle8i Data Cartridge Developer's Guide*.

Issuing a DELETE statement against a table fires any DELETE triggers defined on the table.

All table or index space released by the deleted rows is retained by the table and index.

Restrictions:

- You cannot execute this statement if *table* (or the base table of *view*) contains any domain indexes marked LOADING or FAILED.
- You cannot specify the *sample_clause* in a DELETE statement.
- You cannot specify the ORDER BY clause in the subquery of the *table_expression_clause*.
- You cannot delete from a view except through INSTEAD OF triggers if the view's defining query contains one of the following constructs:

  - A set operator

  - A DISTINCT operator

  - An aggregate function

  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause

  - A collection expression in a SELECT list

  - A subquery in a SELECT list

  - Joins (with some exceptions). See *Oracle8i Administrator's Guide* for details.

- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, the DELETE statement will fail unless the SKIP_UNUSABLE_INDEXES parameter has been set to TRUE. For more information, see "ALTER SESSION" on page 7-78.

PARTITION *(partition_name)* | SUBPARTITION *(subpartition_name)*	specifies that *partition_name* or *subpartition_name* is the name of the partition or subpartition within *table* targeted for deletes. You need not specify the partition name when deleting values from a partitioned table. However, in some cases specifying the partition name is more efficient than a complicated *where_clause*.
dblink	is the complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see "Referring to Objects in Remote Databases" on page 2-74. You can delete rows from a remote table or view only if you are using Oracle's distributed functionality. If you omit *dblink*, Oracle assumes that the table or view is located on the local database.
with_clause	restricts the subquery in one of the following ways: ■ WITH READ ONLY specifies that the subquery cannot be updated. ■ WITH CHECK OPTION specifies that Oracle prohibits any changes to that table that would produce rows that are not included in the subquery. See the WITH CHECK OPTION Example on page 7-558.
table_collection_expression	informs Oracle that the collection value expression should be treated as a table. You can use a *table_collection_expression* to delete only those rows that also exist in another table. *collection_expression* is a subquery that selects a nested table column from *table* or *view*. **Note:** In earlier releases of Oracle, *table_collection_expression* was expressed as "THE subquery". That usage is now deprecated.
where_clause	deletes only rows that satisfy the condition. The condition can reference the table and can contain a subquery. See the syntax description in "Conditions" on page 5-13. You can delete rows from a remote table or view only if you are using Oracle's distributed functionality. If you omit *dblink*, Oracle assumes that the table or view is located on the local database. If you omit the *where_clause*, Oracle deletes all rows of the table or view.
t_alias	provides a **correlation name** for the table, view, subquery, or collection value to be referenced elsewhere in the statement. Table aliases are generally used in DELETE statements with correlated queries. **Note:** This alias is required if the *table_expression_clause* references any object type attributes or object type methods.
returning_clause	retrieves the rows affected by the DELETE statement.

You can use a *returning_clause* to return values from deleted columns, and thereby eliminate the need to issue a SELECT statement following the DELETE statement.

- When deleting a single row, a DELETE statement with a *returning_clause* can retrieve column expressions using the deleted row, rowid, and REFs to the deleted row and store them in PL/SQL variables or bind variables.
- When deleting multiple rows, a DELETE statement with the *returning_clause* stores values from expressions, rowids, and REFs involving the deleted rows in bind arrays.

You can also use DELETE with a *returning_clause* to delete from views with single base tables.

For host binds, the datatype and size of the expression must be compatible with the bind variable.

expr is any of the syntax descriptions in "Expressions" on page 5-1. You must specify a column expression in the *returning_clause* for each variable in the *data_item* list.

INTO indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item* list.

data_item is a PL/SQL variable or bind variable that stores the retrieved *expr* value.

Restrictions:

- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.

Examples

Basic Examples The following statement deletes all rows from a table named TEMP_ASSIGN.

```
DELETE FROM temp_assign;
```

The following statement deletes from the EMP table all sales staff who made less than $100 commission last month:

```
DELETE FROM emp
    WHERE JOB = 'SALESMAN'
    AND COMM < 100;
```

The following statement has the same effect as the preceding example, but uses a subquery:

```
DELETE FROM (select * from emp)
    WHERE JOB = 'SALESMAN'
    AND COMM < 100;
```

Remote Database Example The following statement deletes all rows from the bank account table owned by the user BLAKE on a database accessible by the database link DALLAS:

```
DELETE FROM blake.accounts@dallas;
```

Nested Table Example The following example deletes rows of nested table PROJS where the department number is either 123 or 456, or the department's budget is greater than 456.78:

```
DELETE THE(SELECT projs
           FROM dept d WHERE d.dno = 123) AS p
   WHERE p.pno IN (123, 456) OR p.budgets > 456.78;
```

Partition Example The following example removes rows from partition NOV98 of the SALES table:

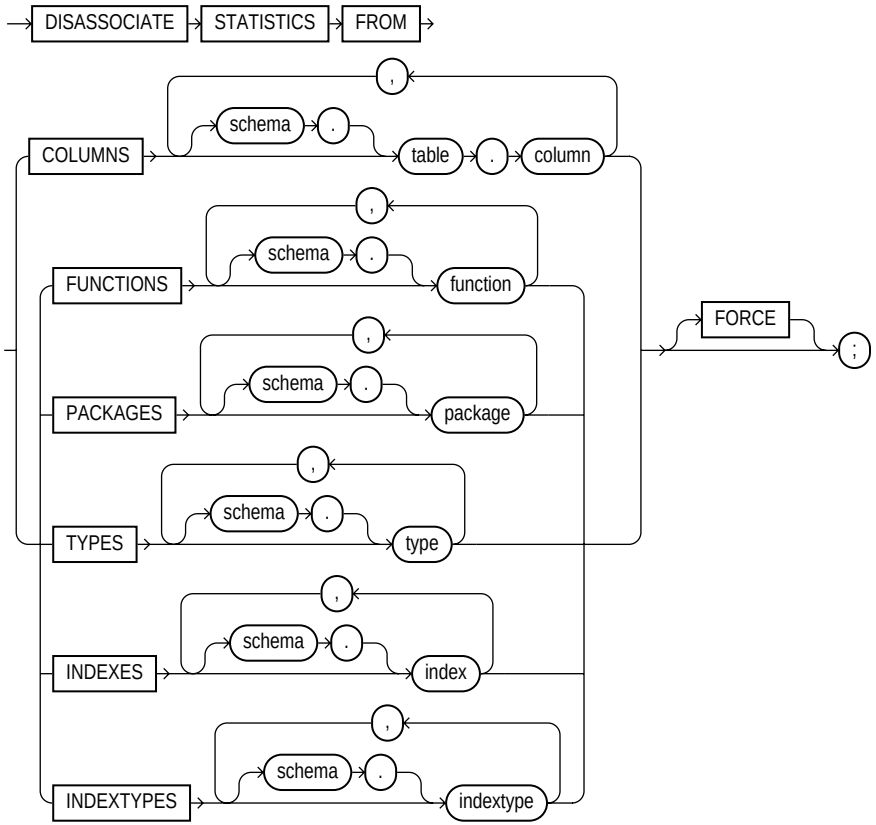
```
DELETE FROM sales PARTITION (nov98)
   WHERE amount_of_sale != 0;
```

Example The following example returns column SAL from the deleted rows and stores the result in bind array :1:

```
DELETE FROM emp
   WHERE job = 'SALESMAN' AND COMM < 100
   RETURNING sal INTO :1;
```

DISASSOCIATE STATISTICS

Syntax



Purpose

To disassociate a statistics type (or default statistics) from columns, standalone functions, packages, types, domain indexes, or indextypes.

For more information on statistics type associations, see "ASSOCIATE STATISTICS" on page 7-194.

Prerequisites

To issue this statement, you must have the appropriate privileges to alter the base object (table, function, package, type, domain index, or indextype).

Keywords and Parameters

FROM COLUMNS | FUNCTIONS | PACKAGES | TYPES | INDEXES | INDEXTYPES	specifies a list of columns, standalone functions, packages, types, domain indexes, or indextypes from which you are disassociating statistics. If you do not specify *schema*, Oracle assumes the object is in your own schema. If you have collected user-defined statistics on the object, the statement fails unless you specify FORCE.
FORCE	deletes the association regardless of whether any statistics exist for the object using the statistics type. If statistics do exist, the statistics are deleted before the association is deleted. **Note:** When you drop an object with which a statistics type has been associated, Oracle automatically disassociates the statistics type with the FORCE option and drops all statistics that have been collected with the statistics type.

Example

This statement disassociates statistics from the PACK package in the HR schema:

```
DISASSOCIATE STATISTICS FROM PACKAGES hr.pack;
```

DROP CLUSTER

Syntax

Purpose

To remove a cluster from the database.

You cannot uncluster an individual table. Instead you must

- Create a new table with the same structure and contents as the old one, but with no CLUSTER clause,
- Drop the old table,
- Use the RENAME statement to give the new table the name of the old one, and
- Grant privileges on the new unclustered table, as grants on the old clustered table do not apply.

See "CREATE TABLE" on page 7-359, "DROP TABLE" on page 7-475, "RENAME" on page 7-527, "GRANT system_privileges_and_roles" on page 7-493.

Prerequisites

The cluster must be in your own schema or you must have the DROP ANY CLUSTER system privilege.

Keywords and Parameters

schema	is the schema containing the cluster. If you omit *schema*, Oracle assumes the cluster is in your own schema.
cluster	is the name of the cluster to be dropped. Dropping a cluster also drops the cluster index and returns all cluster space, including data blocks for the index, to the appropriate tablespace(s).

INCLUDING TABLES	drops all tables that belong to the cluster.
CASCADE CONSTRAINTS	drops all referential integrity constraints from tables outside the cluster that refer to primary and unique keys in tables of the cluster. If you omit this clause and such referential integrity constraints exist, Oracle returns an error and does not drop the cluster.

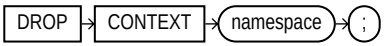
Example

This statement drops a cluster named GEOGRAPHY, all its tables, and any referential integrity constraints that refer to primary or unique keys in those tables:

```
DROP CLUSTER geography
    INCLUDING TABLES
    CASCADE CONSTRAINTS;
```

DROP CONTEXT

Syntax



Purpose

To remove a context namespace from the database. For more information on contexts, see "CREATE CONTEXT" on page 7-243 and *Oracle8i Concepts*.

> **Note:** Removing a context namespace does not invalidate any context under that namespace that has been set for a user session. However, the context will be invalid the next time the user attempts to set that context.

Prerequisites

You must have the DROP ANY CONTEXT system privilege.

Keywords and Parameters

namespace	is the name of the context namespace to drop. You cannot drop the build-in namespace USERENV.

DROP DATABASE LINK

Syntax

Purpose

To remove a database link from the database.

Prerequisites

To drop a private database link, the database link must be in your own schema. To drop a PUBLIC database link, you must have the DROP PUBLIC DATABASE LINK system privilege.

For information on creating database links, see "CREATE DATABASE LINK" on page 7-255.

Keywords and Parameters

PUBLIC	must be specified to drop a PUBLIC database link.
dblink	specifies the database link to be dropped. **Restriction:** You cannot drop a database link in another user's schema and you cannot qualify *dblink* with the name of a schema. (Periods are permitted in names of database links. Therefore, Oracle interprets the entire name, such as RALPH.LINKTOSALES, as the name of a database link in your schema rather than as a database link named LINKTOSALES in the schema RALPH.)

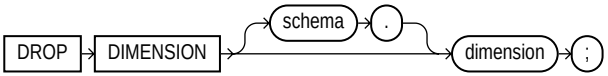
Example

The following statement drops a private database link named BOSTON:

```
DROP DATABASE LINK boston;
```

DROP DIMENSION

Syntax



Purpose

To remove the named dimension.

For information on materialized views and their use of dimensions, see *Oracle8i Concepts*. See also "CREATE DIMENSION" on page 7-259.

Prerequisites

The dimension must be in your own schema or you must have the DROP ANY DIMENSION system privilege to use this statement.

Keywords and Parameters

schema	is the name of the schema in which the dimension is located. If you omit *schema*, Oracle assumes the dimension is in your own schema.
dimension	is the name of the dimension you want to drop. The dimension must already exist. This statement does not invalidate materialized views that use relationships specified in dimension. However, requests that have been rewritten by query rewrite may be invalidated, and subsequent operations on such views may execute more slowly.

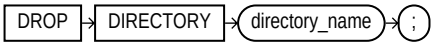
Example

This example drops the TIME dimension:

```
DROP DIMENSION time;
```

DROP DIRECTORY

Syntax



Purpose

Use DROP DIRECTORY to remove a directory object from the database.

For information on creating a directory, see "CREATE DIRECTORY" on page 7-264.

Prerequisites

To drop a directory you must have the DROP ANY DIRECTORY system privilege.

WARNING: Do not drop a directory when files in the associated file system are being accessed by PL/SQL or OCI programs.

Keywords and Parameters

directory_name	is the name of the directory database object to be dropped. Oracle removes the directory object, but does not delete the associated operating system directory on the server's file system.

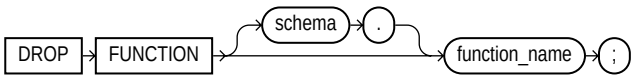
Example

The following statement drops the directory object BFILE_DIR:

```
DROP DIRECTORY bfile_dir;
```

DROP FUNCTION

Syntax



Purpose

To remove a standalone stored function from the database. For information on creating a function, see "CREATE FUNCTION" on page 7-266.

> **Note:** Do not use this statement to remove a function that is part of a package. Instead, either drop the entire package using the DROP PACKAGE statement or redefine the package without the function using the CREATE PACKAGE statement with the OR REPLACE clause.

Prerequisites

The function must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

Keywords and Parameters

schema	is the schema containing the function. If you omit *schema*, Oracle assumes the function is in your own schema.
function_name	is the name of the function to be dropped. Oracle invalidates any local objects that depend on, or call, the dropped function. If you subsequently reference one of these objects, Oracle tries to recompile the object and returns an error if you have not re-created the dropped function. For more information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8i Concepts*. If any statistics types are associated with the function, Oracle disassociates the statistics types with the FORCE option and drops any user-defined statistics collected with the statistics type. For more information on statistics type associations, see "ASSOCIATE STATISTICS" on page 7-194 and "DISASSOCIATE STATISTICS" on page 7-444.

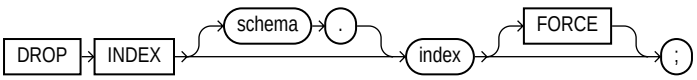
Example

The following statement drops the function NEW_ACCT in the schema RIDDLEY and invalidates all objects that depend upon NEW_ACCT:

```
DROP FUNCTION riddley.new_acct;
```

DROP INDEX

Syntax



Purpose

To remove an index or domain index from the database.

For more information on indexes, see "CREATE INDEX" on page 7-273 and "ALTER INDEX" on page 7-29. For more information on domain indexes, see the *domain_index_clause* of "CREATE INDEX" on page 7-273.

Prerequisites

The index must be in your own schema or you must have the DROP ANY INDEX system privilege.

Keywords and Parameters

schema is the schema containing the index. If you omit *schema*, Oracle assumes the index is in your own schema.

index is the name of the index to be dropped. When the index is dropped, all data blocks allocated to the index are returned to the index's tablespace.

If you drop a **domain index**:

- Oracle invokes the appropriate indextype drop routine. For information on these routines, see *Oracle8i Data Cartridge Developer's Guide*.
- In addition, if any statistics are associated with the domain index, Oracle disassociates the statistics types with the FORCE clause and removes the user-defined statistics collected with the statistics type. For more information on statistics type associations, see "ASSOCIATE STATISTICS" on page 7-194 and "DISASSOCIATE STATISTICS" on page 7-444.

If you drop a global partitioned index, a range-partitioned, or a hash-partitioned index, all the index partitions are also dropped. If you drop a a composite-partitioned index, all the index partitions and subpartitions are also dropped.

FORCE	applies only to domain indexes. This clause drops the domain index even if the indextype routine invocation returns an error or the index is marked LOADING. Without FORCE, you cannot drop a domain index if its indextype routine invocation returns an error or the index is marked LOADING.

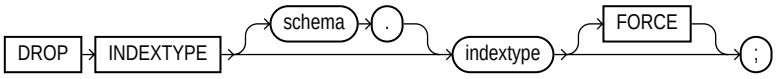
Example

This statement drops an index named MONOLITH:

```
DROP INDEX monolith;
```

DROP INDEXTYPE

Syntax



Purpose

To drop an indextype, as well as any association with a statistics type.

For more information on indextypes, see "CREATE INDEXTYPE" on page 7-291.

Prerequisites

The indextype must be in your own schema or you must have the DROP ANY INDEXTYPE system privilege.

Keywords and Parameters

schema	is the schema containing the indextype. If you omit *schema*, Oracle assumes the indextype is in your own schema.
indextype	is the name of the indextype to be dropped. If any statistics types have been associated with indextype, Oracle disassociates the statistics type from the indextype and drops any statistics that have been collected using the statistics type. For more information on statistics associations, see "ASSOCIATE STATISTICS" on page 7-194 and "DISASSOCIATE STATISTICS" on page 7-444.
FORCE	drops the indextype even if the indextype is currently being referenced by one or more domain indexes, and marks those domain indexes INVALID. Without FORCE, you cannot drop an indextype if any domain indexes reference the indextype.

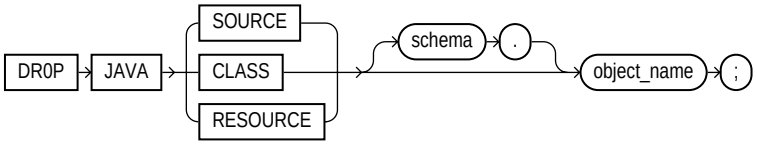
Example

The following statement drops the indextype TEXTINDEXTYPE and marks INVALID any domain indexes defined on this indextype:

```
DROP INDEXTYPE textindextype FORCE;
```

DROP JAVA

Syntax



Purpose

To drop a Java source, class, or resource schema object.

For more information on resolving Java sources, classes, and resources, see *Oracle8i Java Stored Procedures Developer's Guide.*

Prerequisites

The Java source, class, or resource must be in your own schema or you must have the DROP ANY PROCEDURE system privilege. You also must have the EXECUTE object privilege on Java classes to use this command.

Keywords and Parameters

JAVA SOURCE	drops a Java source schema object and all Java class schema objects derived from it.
JAVA CLASS	drops a Java class schema object.
JAVA RESOURCE	drops a Java resource schema object.
object_name	specifies the name of an existing Java class, source, or resource schema object.

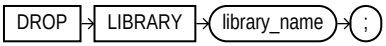
Example

The following statement drops the Java class `MyClass`:

```
DROP JAVA CLASS "MyClass";
```

DROP LIBRARY

Syntax



Purpose

To remove an external procedure library from the database.

For information on creating a library, see "CREATE LIBRARY" on page 7-298.

Prerequisites

You must have the DROP LIBRARY system privilege.

Keywords and Parameters

library_name	is the name of the external procedure library being dropped.

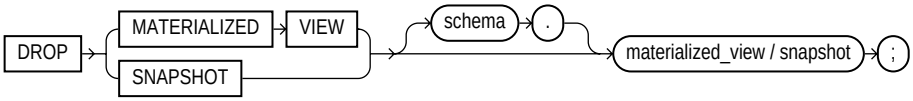
Example

The following statement drops the EXT_PROCS library:

```
DROP LIBRARY ext_procs;
```

DROP MATERIALIZED VIEW / SNAPSHOT

Syntax



Purpose

To remove an existing materialized view from the database.

The terms "snapshot" and "materialized view" are synonymous. For more information on materialized views, including a description of the various types of materialized views, see "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 7-300. For information on materialized views in a replication environment, see *Oracle8i Replication*. For information on materialized views in a data warehousing environment, see *Oracle8i Tuning*.

Prerequisites

The materialized view must be in your own schema or you must have the DROP ANY MATERIALIZED VIEW (or DROP ANY SNAPSHOT) system privilege. You must also have the privileges to drop the internal table, views, and index that Oracle uses to maintain the materialized view's data.

For information on these privileges, see "DROP TABLE" on page 7-475, "DROP VIEW" on page 7-485, and "DROP INDEX" on page 7-454.

Keywords and Parameters

schema is the schema containing the materialized view. If you omit *schema*, Oracle assumes the materialized view is in your own schema.

materialized view / snapshot is the name of the existing materialized view to be dropped.

- If you drop a simple materialized view that is the least recently refreshed materialized view of a master table, Oracle automatically purges from the detail table's materialized view log only the rows needed to refresh the dropped materialized view.

- If you drop a detail table, Oracle does not automatically drop materialized views based on the table. However, Oracle returns an error when it tries to refresh a materialized view based on a detail table that has been dropped.
- If you drop a materialized view, any compiled requests that were rewritten to use the materialized view will be invalidated and recompiled automatically. If the materialized view was prebuilt on a table, the table is not dropped, but it can no longer be maintained by the materialized view refresh mechanism.

Examples

The following statement drops the materialized view PARTS owned by the user HQ:

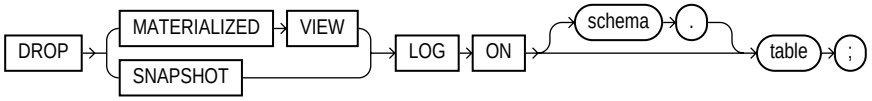
```
DROP SNAPSHOT hq.parts;
```

The following statement drops the SALES_BY_MONTH materialized view and the underlying table of the materialized view (unless the underlying table was registered in the CREATE MATERIALIZED VIEW statement with the ON PREBUILT TABLE clause):

```
DROP MATERIALIZED VIEW sales_by_month;
```

DROP MATERIALIZED VIEW LOG / SNAPSHOT LOG

Syntax



Purpose

To remove a materialized view log from the database.

The terms "snapshot" and "materialized view" are synonymous. For more information on materialized views, including a description of the various types of materialized views and refreshing materialized views, see "CREATE MATERIALIZED VIEW / SNAPSHOT" on page 7-300 and "ALTER MATERIALIZED VIEW / SNAPSHOT" on page 7-45.

For information on materialized view logs, see "CREATE MATERIALIZED VIEW LOG / SNAPSHOT LOG" on page 7-314.

For information on materialized views in a replication environment, see *Oracle8i Replication*. For information on materialized views in a data warehousing environment, see *Oracle8i Tuning*.

Prerequisites

A materialized view log consists of a table and a trigger. To drop a materialized view log, you must have the privileges listed for "DROP TABLE" on page 7-475.

Keywords and Parameters

schema	is the schema containing the materialized view log and its master table. If you omit *schema*, Oracle assumes the materialized view log and master table are in your own schema.
table	is the name of the detail table associated with the materialized view log to be dropped.

After you drop a materialized view log, some materialized views based on the materialized view log's detail table can no longer be fast refreshed. These materialized views include rowid materialized views, primary key materialized views, and subquery materialized views. For a description of the types of materialized views, see *Oracle8i Tuning*.

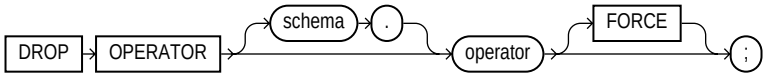
Example

The following statement drops the materialized view log on the PARTS master table:

```
DROP MATERIALIZED VIEW LOG ON parts;
```

DROP OPERATOR

Syntax



Purpose

To drop a user-defined operator.

For more information on operators, see "User-Defined Operators" on page 3-16, *Oracle8i Data Cartridge Developer's Guide* and "CREATE OPERATOR" on page 7-320.

Prerequisites

The operator must be in your schema or you must have the DROP ANY OPERATOR system privilege.

Keywords and Parameters

schema	is the schema containing the operator. If you omit *schema*, Oracle assumes the operator is in your own schema.
operator	specifies the name of the operator to be dropped.
FORCE	drops the operator even if it is currently being referenced by one or more schema objects (indextypes, packages, functions, procedures, and so on), and marks those dependent objects INVALID. Without FORCE, you cannot drop an operator if any schema objects reference it.

Example

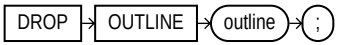
The following statement drops the operator MERGE:

```
DROP OPERATOR ordsys.merge;
```

Because the FORCE clause is not specified, this operation will fail if any of the bindings of this operator are referenced by an indextype.

DROP OUTLINE

Syntax



Purpose

To drop a stored outline.

For more information on outlines, see "CREATE OUTLINE" on page 7-323 and *Oracle8i Tuning*.

Prerequisites

To drop an outline, you must have the DROP ANY OUTLINE system privilege.

Keywords and Parameters

outline	is the name of the outline to be dropped. After the outline is dropped, if the SQL statement for which the stored outline was created is compiled, the optimizer generates a new execution plan without the influence of the outline.

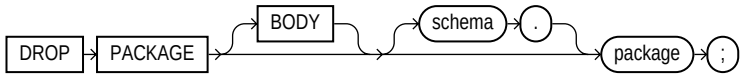
Example

The following statement drops the stored outline called SALARIES.

```
DROP OUTLINE salaries;
```

DROP PACKAGE

Syntax



Purpose

To remove a stored package from the database. This statement drops the body and specification of a package.

> **Note:** Do not use this statement to remove a single object from a package. Instead, re-create the package without the object using the CREATE PACKAGE and CREATE PACKAGE BODY statements with the OR REPLACE clause. See "CREATE PACKAGE" on page 7-325.

Prerequisites

The package must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

Keywords and Parameters

BODY	drops only the body of the package. If you omit this clause, Oracle drops both the body and specification of the package. When you drop only the body of a package but not its specification, Oracle does not invalidate dependent objects. However, you cannot call one of the procedures or stored functions declared in the package specification until you re-create the package body.
schema	is the schema containing the package. If you omit *schema*, Oracle assumes the package is in your own schema.

package	is the name of the package to be dropped. Oracle invalidates any local objects that depend on the package specification. If you subsequently reference one of these objects, Oracle tries to recompile the object and returns an error if you have not re-created the dropped package. For information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8i Concepts*. If any statistics types are associated with the package, Oracle disassociates the statistics types with the FORCE clause and drops any user-defined statistics collected with the statistics types. For more information, see "ASSOCIATE STATISTICS" on page 7-194 and "DISASSOCIATE STATISTICS" on page 7-444.

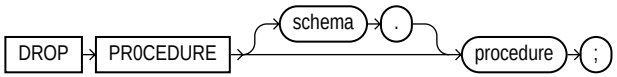
Example

The following statement drops the specification and body of the BANKING package, invalidating all objects that depend on the specification:

```
DROP PACKAGE banking;
```

DROP PROCEDURE

Syntax



Purpose

To remove a standalone stored procedure from the database. Do not use this statement to remove a procedure that is part of a package. Instead, either drop the entire package using the DROP PACKAGE statement, or redefine the package without the procedure using the CREATE PACKAGE statement with the OR REPLACE clause.

For information on creating a procedure, see "CREATE PROCEDURE" on page 7-333.

Prerequisites

The procedure must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

Keywords and Parameters

schema	is the schema containing the procedure. If you omit *schema*, Oracle assumes the procedure is in your own schema.
procedure	is the name of the procedure to be dropped. When you drop a procedure, Oracle invalidates any local objects that depend upon the dropped procedure. If you subsequently reference one of these objects, Oracle tries to recompile the object and returns an error message if you have not re-created the dropped procedure. For information on how Oracle maintains dependencies among schema objects, including remote objects, see *Oracle8i Concepts*.

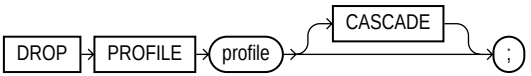
Example

The following statement drops the procedure TRANSFER owned by the user KERNER and invalidates all objects that depend upon TRANSFER:

```
DROP PROCEDURE kerner.transfer
```

DROP PROFILE

Syntax



Purpose

To remove a profile from the database.

For information on creating a profile, see "CREATE PROFILE" on page 7-338.

Prerequisites

You must have the DROP PROFILE system privilege.

Keywords and Parameters

profile	is the name of the profile to be dropped. **Restriction:** You cannot drop the DEFAULT profile.
CASCADE	deassigns the profile from any users to whom it is assigned. Oracle automatically assigns the DEFAULT profile to such users. You must specify this clause to drop a profile that is currently assigned to users.

Example

The following statement drops the profile ENGINEER:

```
DROP PROFILE engineer CASCADE;
```

Oracle drops the profile ENGINEER and assigns the DEFAULT profile to any users currently assigned the ENGINEER profile.

DROP ROLE

Syntax

Purpose

To remove a role from the database. When you drop a role, Oracle revokes it from all users and roles to whom it has been granted and removes it from the database.

For information on creating roles, see "CREATE ROLE" on page 7-344. For information on disabling roles for the current session, see "SET ROLE" on page 7-570.

Prerequisites

You must have been granted the role with the ADMIN OPTION or you must have the DROP ANY ROLE system privilege.

Keywords and Parameters

role	is the role to be dropped.

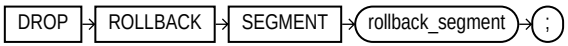
Example

To drop the role FLORIST, issue the following statement:

```
DROP ROLE florist;
```

DROP ROLLBACK SEGMENT

Syntax



Purpose

To remove a rollback segment from the database. When you drop a rollback segment, all space allocated to the rollback segment returns to the tablespace.

For information on creating a rollback segment, see CREATE ROLLBACK SEGMENT on page 7-346. See also "CREATE TABLESPACE" on page 7-394.

Prerequisites

You must have the DROP ROLLBACK SEGMENT system privilege.

Keywords and Parameters

rollback_segment is the name the rollback segment to be dropped.

Restrictions:

- You can drop a rollback segment only if it is offline. To determine whether a rollback segment is offline, query the data dictionary view DBA_ROLLBACK_SEGS. Offline rollback segments have the value AVAILABLE in the STATUS column. You can take a rollback segment offline with the OFFLINE clause of the ALTER ROLLBACK SEGMENT statement.
- You cannot drop the SYSTEM rollback segment.

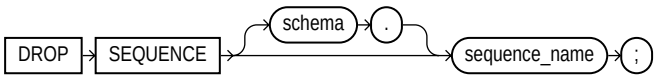
Example

The following statement drops the rollback segment ACCOUNTING:

```
DROP ROLLBACK SEGMENT accounting;
```

DROP SEQUENCE

Syntax



Purpose

To remove a sequence from the database.

You can also use this statement to restart a sequence by dropping and then re-creating it. For example, if you have a sequence with a current value of 150 and you would like to restart the sequence with a value of 27, you can drop the sequence and then re-create it with the same name and a START WITH value of 27.

For more information on creating and modifying sequences, see "CREATE SEQUENCE" on page 7-350 and "ALTER SEQUENCE" on page 7-76.

Prerequisites

The sequence must be in your own schema or you must have the DROP ANY SEQUENCE system privilege.

Keywords and Parameters

schema	is the schema containing the sequence. If you omit *schema*, Oracle assumes the sequence is in your own schema.
sequence_name	is the name of the sequence to be dropped.

Example

The following statement drops the sequence ESEQ owned by the user ELLY. To issue this statement, you must either be connected as user ELLY or have DROP ANY SEQUENCE system privilege:

```
DROP SEQUENCE elly.eseq;
```

DROP SNAPSHOT

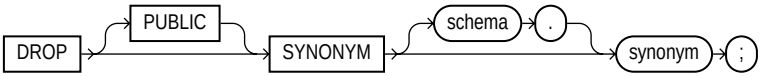
In Oracle8*i*, "snapshots" are synonymous with "materialized views." Please see "DROP MATERIALIZED VIEW / SNAPSHOT" on page 7-459.

DROP SNAPSHOT LOG

In Oracle8*i*, "snapshots" are synonymous with "materialized views." Please see "DROP MATERIALIZED VIEW LOG / SNAPSHOT LOG" on page 7-461.

DROP SYNONYM

Syntax



Purpose

To remove a synonym from the database, or to change the definition of a synonym by dropping and re-creating it.

For more information on synonyms, see "CREATE SYNONYM" on page 7-356.

Prerequisites

To drop a private synonym, either the synonym must be in your own schema or you must have the DROP ANY SYNONYM system privilege.

To drop a PUBLIC synonym, either the synonym must be in your own schema or you must have the DROP ANY PUBLIC SYNONYM system privilege.

Keywords and Parameters

PUBLIC	must be specified to drop a public synonym. You cannot specify *schema* if you have specified PUBLIC.
schema	is the schema containing the synonym. If you omit *schema*, Oracle assumes the synonym is in your own schema.
synonym	is the name of the synonym to be dropped. If you drop a synonym for a materialized view, or its containing table or snapshot, or any of its dependent tables, the materialized view will be invalidated.

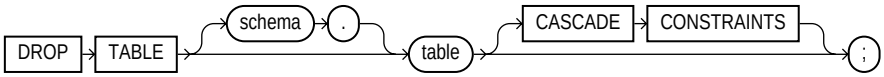
Example

To drop a synonym named MARKET, issue the following statement:

```
DROP SYNONYM market;
```

DROP TABLE

Syntax



Purpose

To remove a table or an object table and all its data from the database.

For information on creating tables, see "CREATE TABLE" on page 7-359. For information on modifying tables, see "ALTER TABLE" on page 7-113.

Prerequisites

The table must be in your own schema or you must have the DROP ANY TABLE system privilege.

Keywords and Parameters

schema is the schema containing the table. If you omit *schema*, Oracle assumes the table is in your own schema.

table is the name of the table, object table, or index-organized table to be dropped. Oracle automatically performs the following operations:

- Removes all rows from the table (as if the rows were deleted).
- Drops all the table's indexes and domain indexes, regardless of who created them or whose schema contains them.
- If you drop a range-partitioned or hash-partitioned table, all the table partitions are also dropped. If you drop a composite-partitioned table, all the partitions and subpartitions are also dropped.
- For a domain index, this statement invokes the appropriate drop routines. For more information on these routines, see *Oracle8i Data Cartridge Developer's Guide*.
- If any statistic types are associated with the table, Oracle disassociates the statistics types with the FORCE clause and removes any user-defined statistics collected with the statistics type. For more information on statistics type associations, see "ASSOCIATE STATISTICS" on page 7-194 and "DISASSOCIATE STATISTICS" on page 7-444.

	■ If the table is not part of a cluster, Oracle returns all data blocks allocated to the table and its indexes to the tablespaces containing the table and its indexes. ■ If the table is a base table for a view, a container or master table of a materialized view, or if it is referenced in a stored procedure, function, or package, Oracle invalidates these dependent objects but does not drop them. You cannot use these objects unless you re-create the table or drop and re-create the objects so that they no longer depend on the table. ■ If you choose to re-create the table, it must contain all the columns selected by the queries originally used to define the materialized views/snapshots and all the columns referenced in the stored procedures, functions, or packages. Any users previously granted object privileges on the views, stored procedures, functions, or packages need not be regranted these privileges. ■ If the table is a detail table for a materialized view, the materialized view can still be queried, but it cannot be refreshed unless the table is re-created so that it contains all the columns selected by the materialized view's query. ■ If the table has a materialized view log/snapshot log, Oracle drops this log and any other direct-load INSERT refresh information associated with the table. **Note:** To drop a cluster and all its the tables, use the DROP CLUSTER statement with the INCLUDING TABLES clause to avoid dropping each table individually. See "DROP CLUSTER" on page 7-446.
CASCADE CONSTRAINTS	drops all referential integrity constraints that refer to primary and unique keys in the dropped table. If you omit this clause, and such referential integrity constraints exist, Oracle returns an error and does not drop the table.

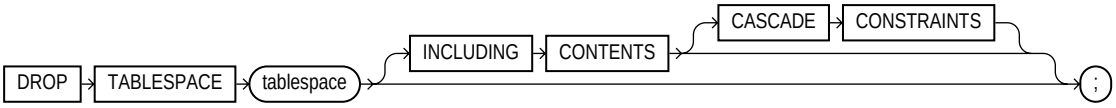
Example

The following statement drops the TEST_DATA table:

```
DROP TABLE test_data;
```

DROP TABLESPACE

Syntax



Purpose

To remove a tablespace from the database.

For information on creating and modifying tablespaces, see "CREATE TABLESPACE" on page 7-394 and "ALTER TABLESPACE" on page 7-164.

Prerequisites

You must have the DROP TABLESPACE system privilege. You cannot drop a tablespace if it contains any rollback segments holding active transactions.

Keywords and parameters

tablespace is the name of the tablespace to be dropped.

You can drop a tablespace regardless of whether it is online or offline. Oracle recommends that you take the tablespace offline before dropping it to ensure that no SQL statements in currently running transactions access any of the objects in the tablespace.

You may want to alert any users who have been assigned the tablespace as either a default or temporary tablespace. After the tablespace has been dropped, these users cannot allocate space for objects or sort areas in the tablespace. You can reassign users new default and temporary tablespaces with the ALTER USER statement.

Restrictions:

- You cannot drop the SYSTEM tablespace.
- You cannot drop a tablespace that contains a domain index or any objects created by a domain index. For more information on domain indexes, see *Oracle8i Data Cartridge Developer's Guide* and *Oracle8i Concepts*.

INCLUDING CONTENTS drops all the contents of the tablespace. You must specify this clause to drop a tablespace that contains any database objects. If you omit this clause, and the tablespace is not empty, Oracle returns an error and does not drop the tablespace.

For **partitioned tables**, DROP TABLESPACE will fail even if you specify INCLUDING CONTENTS, if the tablespace contains some, but not all,

- partitions of a range- or hash-partitioned table, or
- subpartitions of a composite-partitioned table.

  Note: If all the partitions of a partitioned table reside in *tablespace*, DROP TABLESPACE ... INCLUDING CONTENTS will drop *tablespace*, as well as any associated index segments, LOB data segments, and LOB index segments in the other tablespace(s).

For a **partitioned index-organized table**, if all the primary key index segments are in this tablespace, this clause will also drop any overflow segments that exist in other tablespaces. If some of the primary key index segments are *not* in this tablespace, the statement will fail. In that case, before you can drop the tablespace, you must use ALTER TABLE ... MOVE PARTITION to move those primary key index segments into this tablespace, drop the partitions whose overflow data segments are not in this tablespace, and drop the partitioned index-organized table.

If the tablespace contains a container table or detail table of a materialized view, Oracle invalidates the materialized view.

If the tablespace contains a materialized view / snapshot log, Oracle drops this log and any other direct-load INSERT refresh information associated with the table.

CASCADE CONSTRAINTS — drops all referential integrity constraints from tables outside *tablespace* that refer to primary and unique keys of tables inside *tablespace*. If you omit this clause and such referential integrity constraints exist, Oracle returns an error and does not drop the tablespace.

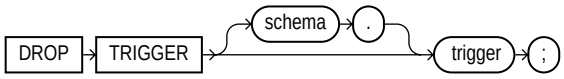
Example

The following statement drops the MFRG tablespace and all its contents:

```
DROP TABLESPACE mfrg
    INCLUDING CONTENTS
        CASCADE CONSTRAINTS;
```

DROP TRIGGER

Syntax



Purpose

To remove a database trigger from the database.

For information on creating triggers, see "CREATE TRIGGER" on page 7-401.

Prerequisites

The trigger must be in your own schema or you must have the DROP ANY TRIGGER system privilege.

In addition, to drop a trigger on DATABASE in another user's schema, you must have the ADMINISTER DATABASE TRIGGER system privilege. For more information on database triggers, see "CREATE TRIGGER" on page 7-401.

Keywords and Parameters

schema	is the schema containing the trigger. If you omit *schema*, Oracle assumes the trigger is in your own schema.
trigger	is the name of the trigger to be dropped. Oracle removes it from the database and does not fire it again.

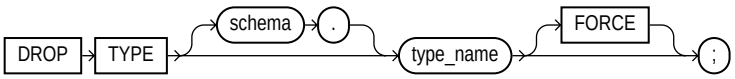
Example

The following statement drops the REORDER trigger in the schema RUTH:

```
DROP TRIGGER ruth.reorder;
```

DROP TYPE

Syntax



Purpose

To drop the specification and body of an object, a varray, or nested table type. To drop just the body of an object type, see "DROP TYPE BODY" on page 7-482.

For more information on types, see "CREATE TYPE" on page 7-411.

Prerequisites

The object, varray, or nested table type must be in your own schema or you must have the DROP ANY TYPE system privilege.

Keywords and Parameters

schema is the schema containing the type. If you omit *schema*, Oracle assumes the type is in your own schema.

type_name is the name of the object, varray, or nested table type to be dropped. You can drop only types with no type or table dependencies.

If *type_name* is a statistics type, this statement will fail unless you also specify FORCE. If you specify FORCE, Oracle first disassociates all objects that are associated with *type_name*, and then drops *type_name*. For more information on statistics types, see "ASSOCIATE STATISTICS" on page 7-194 and "DISASSOCIATE STATISTICS" on page 7-444.

If *type_name* is an object type that has been associated with a statistics type, Oracle first attempts to disassociate *type_name* from the statistics type and then drop *type_name*. However, if statistics have been collected using the statistics type, Oracle will be unable to disassociate *type_name* from the statistics type, and this statement will fail.

If *type_name* is an implementation type for an indextype, the indextype will be marked INVALID. For more information, see "CREATE INDEXTYPE" on page 7-291.

Unless you specify FORCE, you can drop only object, nested table, or varray types that are standalone schema objects with no dependencies. This is the default behavior.

FORCE	forces the type to be dropped even if it has dependent database objects. Oracle marks UNUSED all columns dependent on the type to be dropped, and those columns become inaccessible.
	WARNING: Oracle does not recommend that you specify FORCE to drop types with dependencies. This operation is not recoverable and could cause the data in the dependent tables or columns to become inaccessible. For information about type dependencies, see *Oracle8i Application Developer's Guide - Fundamentals***.**

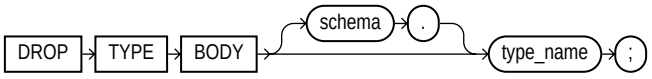
Example

The following statement removes object type PERSON_T:

```
DROP TYPE person_t;
```

DROP TYPE BODY

Syntax



Purpose

To drop the body of an object, varray, or nested table type. When you drop a type body, the object type specification still exists, and you can re-create the type body. Prior to re-creating the body, you can still use the object type, although you cannot call the member functions.

To drop the specification of an object, see "DROP TYPE" on page 7-480. For more information on type bodies, see "CREATE TYPE BODY" on page 7-421.

Prerequisites

The object type body must be in your own schema, and you must have

- the CREATE TYPE or CREATE ANY TYPE system privilege, or
- the DROP ANY TYPE system privilege

Keywords and Parameters

schema	is the schema containing the object type. If you omit *schema*, Oracle assumes the object type is in your own schema.
type_name	is the name of the object type body to be dropped. **Restriction:** You can drop a type body only if it has no type or table dependencies.

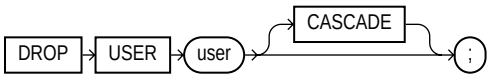
Example

The following statement removes object type body RATIONAL:

```
DROP TYPE BODY rational;
```

DROP USER

Syntax



Purpose

To remove a database user and optionally remove the user's objects. For information on creating a user, see "CREATE USER" on page 7-425. For information on modifying the definition of a user, see "ALTER USER" on page 7-179.

Prerequisites

You must have the DROP USER system privilege.

Keywords and Parameters

user is the user to be dropped. Oracle does not drop users whose schemas contain objects unless you specify CASCADE, or unless you first explicitly drop the user's objects.

CASCADE drops all objects in the user's schema before dropping the user. You must specify this clause to drop a user whose schema contains any objects.

- If the user's schema contains tables, Oracle drops the tables and automatically drops any referential integrity constraints on tables in other schemas that refer to primary and unique keys on these tables.
- If this clause results in tables being dropped, Oracle also drops all domain indexes created on columns of those tables, and invokes appropriate drop routines. For more information on these routines, see *Oracle8i Data Cartridge Developer's Guide*.
- Oracle invalidates, but does not drop, the following objects in **other** schemas: views or synonyms for objects in the dropped user's schema; and stored procedures, functions, or packages that query objects in the dropped user's schema.
- Oracle does not drop materialized views on tables or views in the dropped user's schema, but if you specify CASCADE, the materialized views can no longer be refreshed.
- Oracle drops all triggers in the user's schema.
- Oracle does **not** drop roles created by the user.

WARNING: Oracle also drops with FORCE all types owned by the user. See the FORCE keyword of "DROP TYPE" on page 7-481.

Examples

If user BRADLEY's schema contains no objects, you can drop BRADLEY by issuing the statement:

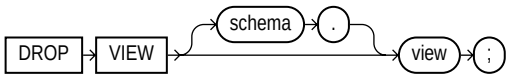
```
DROP USER bradley;
```

If BRADLEY's schema contains objects, you must use the CASCADE clause to drop BRADLEY and the objects:

```
DROP USER bradley CASCADE;
```

DROP VIEW

Syntax



Purpose

To remove a view or an object view from the database. You can change the definition of a view by dropping and re-creating it. For more information, see "CREATE VIEW" on page 7-430.

Prerequisites

The view must be in your own schema or you must have the DROP ANY VIEW system privilege.

Keywords and Parameters

schema	is the schema containing the view. If you omit *schema*, Oracle assumes the view is in your own schema.
view	is the name of the view to be dropped. Views, materialized views, and synonyms that refer to the view are not dropped, but become invalid. You can drop them or redefine views and synonyms, or you can define other views in such a way that the invalid views and synonyms become valid again. See "CREATE TABLE" on page 7-359 and "CREATE SYNONYM" on page 7-356. To revalidate invalid materialized views, see "ALTER MATERIALIZED VIEW / SNAPSHOT" on page 7-45.

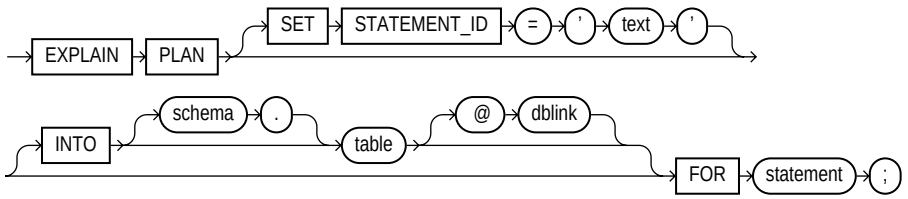
Example

The following statement drops the VIEW_DATA view:

```
DROP VIEW view_data;
```

EXPLAIN PLAN

Syntax



Purpose

To determine the execution plan Oracle follows to execute a specified SQL statement. This statement inserts a row describing each step of the execution plan into a specified table. If you are using cost-based optimization, this statement also determines the cost of executing the statement. If any domain indexes are defined on the table, user-defined CPU and I/O costs will also be inserted. See *Oracle8i Tuning* for information on the output of EXPLAIN PLAN.

The definition of a sample output table PLAN_TABLE is available in a SQL script on your distribution media. Your output table must have the same column names and datatypes as this table. The common name of this script is UTLXPLAN.SQL. The exact name and location depend on your operating system.

> **Note:** Do not use the EXPLAIN PLAN statement to determine the execution plans of SQL statements that access data dictionary views or dynamic performance tables.

You can also issue the EXPLAIN PLAN statement as part of the SQL trace facility. For information on how to use the SQL trace facility, as well as a detailed discussion of how to generate and interpret execution plans, see *Oracle8i Tuning*.

Prerequisites

To issue an EXPLAIN PLAN statement, you must have the privileges necessary to insert rows into an existing output table that you specify to hold the execution plan. For information on these privileges, see "INSERT" on page 7-512.

You must also have the privileges necessary to execute the SQL statement for which you are determining the execution plan. If the SQL statement accesses a view, you must have privileges to access any tables and views on which the view is based. If the view is based on another view that is based on a table, you must have privileges to access both the other view and its underlying table.

To examine the execution plan produced by an EXPLAIN PLAN statement, you must have the privileges necessary to query the output table. For more information on these privileges, see "SELECT and Subqueries" on page 7-541.

The EXPLAIN PLAN statement is a data manipulation language (DML) statement, rather than a data definition language (DDL) statement. Therefore, Oracle does not implicitly commit the changes made by an EXPLAIN PLAN statement. If you want to keep the rows generated by an EXPLAIN PLAN statement in the output table, you must commit the transaction containing the statement.

Keywords and Parameters

SET STATEMENT_ID	specifies the value of the STATEMENT_ID column for the rows of the execution plan in the output table. You can then use this value to identify these rows among others in the output table. Be sure to specify a STATEMENT_ID value if your output table contains rows from many execution plans. If you omit this clause, the STATEMENT_ID value defaults to null.
INTO	specifies name of the output table, and optionally its schema and database. This table must exist before you use the EXPLAIN PLAN statement. If you omit *schema*, Oracle assumes the table is in your own schema. The *dblink* can be a complete or partial name of a database link to a remote Oracle database where the output table is located. For information on referring to database links, see the section, "Referring to Objects in Remote Databases" on page 2-74. You can specify a remote output table only if you are using Oracle's distributed functionality. If you omit *dblink*, Oracle assumes the table is on your local database. If you omit INTO altogether, Oracle assumes an output table named PLAN_TABLE in your own schema on your local database.
FOR *statement*	specifies a SELECT, INSERT, UPDATE, DELETE, CREATE TABLE, or CREATE INDEX statement for which the execution plan is generated.
	Note: If *statement* includes the *parallel_clause*, the resulting execution plan will indicate parallel execution. However, EXPLAIN PLAN actually inserts the statement into the plan table, so that the parallel DML statement you submit is no longer the first DML statement in the transaction. This violates the Oracle restriction of one parallel DML statement per transaction, and the statement will be executed serially. To maintain parallel execution of the statements, you must commit or roll back the EXPLAIN PLAN statement, and then submit the parallel DML statement.

Examples

The following statement determines the execution plan and cost for an UPDATE statement and inserts rows describing the execution plan into the specified OUTPUT table with the STATEMENT_ID value of 'Raise in Chicago':

```
EXPLAIN PLAN
    SET STATEMENT_ID = 'Raise in Chicago'
    INTO output
    FOR UPDATE emp
        SET sal = sal * 1.10
        WHERE deptno =  (SELECT deptno
                  FROM dept
                  WHERE loc = 'CHICAGO');
```

The following SELECT statement queries the OUTPUT table and returns the execution plan and the cost:

```
SELECT LPAD(' ',2*(LEVEL-1))||operation operation, options,
object_name, position
    FROM output
    START WITH id = 0 AND statement_id = 'Raise in Chicago'
    CONNECT BY PRIOR id = parent_id AND
    statement_id = 'Raise in Chicago';
```

The query returns this execution plan:

```
OPERATION             OPTIONS     OBJECT_NAME     POSITION
-------------------------------------------------------------
UPDATE STATEMENT                                         1
    FILTER                                               0
      TABLE ACCESS    FULL        EMP                    1
      TABLE ACCESS    FULL        DEPT                   2
```

The value in the POSITION column of the first row shows that the statement has a cost of 1.

Partitioned Example Assume that STOCKS is a table with eight partitions on a STOCK_NUM column, and that a local prefixed index STOCK_IX on column STOCK_NUM exists. The partition HIGHVALUES are 1000, 2000, 3000, 4000, 5000, 6000, 7000, and 8000.

Consider the query:

```
SELECT * FROM stocks WHERE stock_num BETWEEN 3800 AND :h;
```

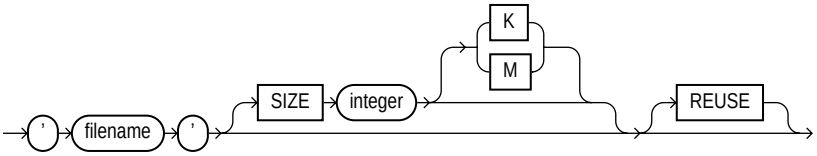
(where :h represents a bind variable). EXPLAIN PLAN executes this query with PLAN_TABLE as the output table. The basic execution plan, including partitioning information, is obtained with the query:

```
SELECT id, operation, options, object_name,
  partition_start, partition_stop, partition_id FROM plan_table;
```

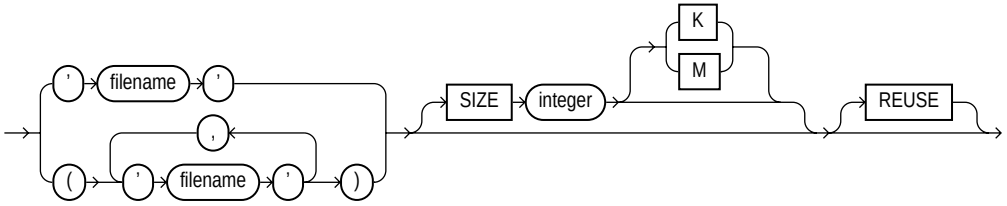
filespec

Syntax

filespec_datafiles & filespec_tempfiles::=



filespec_redo_log_file_groups::=



Purpose

To specify a file as a datafile or tempfile

To specify a group of one or more files as a redo log file group.

Prerequisites

A *filespec* can appear in the following statements: "CREATE DATABASE" on page 7-249, "ALTER DATABASE" on page 7-6, "CREATE TABLESPACE" on page 7-394, and "ALTER TABLESPACE" on page 7-164, "CREATE CONTROLFILE" on page 7-245, "CREATE LIBRARY" on page 7-298, and "CREATE TEMPORARY TABLESPACE" on page 7-399.

You must have the privileges necessary to issue one of these statements.

Keywords and Parameters

'filename'	is the name of either a datafile, tempfile, or a redo log file member. A *'filename'* can contain only single-byte characters from 7-bit ASCII or EBCDIC character sets. Multibyte characters are not valid. A redo log file group can have one or more members (copies). Each *'filename'* must be fully specified according to the conventions for your operating system.
SIZE *integer*	specifies the size of the file. Use K or M to specify the size in kilobytes or megabytes. ■ You can omit this parameter **only** if the file already exists. ■ The size of a tablespace must be one block greater than the sum of the sizes of the objects contained in it.
REUSE	allows Oracle to reuse an existing file. ■ If the file already exists, Oracle verifies that its size matches the value of the SIZE parameter (if you specify SIZE). ■ If the file does not exist, Oracle ignores this clause and creates the file. ■ You can omit this clause only if the file does not already exist. If you omit this clause, Oracle creates the file.
	Note: Whenever Oracle uses an existing file, the file's previous contents are lost.

Examples

The following statement creates a database named PAYABLE that has two redo log file groups, each with two members, and one datafile:

```
CREATE DATABASE payable
   LOGFILE GROUP 1 ('diska:log1.log', 'diskb:log1.log') SIZE 50K,
           GROUP 2 ('diska:log2.log', 'diskb:log2.log') SIZE 50K
   DATAFILE 'diskc:dbone.dat' SIZE 30M;
```

The first *filespec* in the LOGFILE clause specifies a redo log file group with the GROUP value 1. This group has members named 'DISKA:LOG1.LOG' and 'DISKB:LOG1.LOG', each 50 kilobytes in size.

The second *filespec* in the LOGFILE clause specifies a redo log file group with the GROUP value 2. This group has members named 'DISKA:LOG2.LOG' and 'DISKB:LOG2.LOG', also 50 kilobytes in size.

The *filespec* in the DATAFILE clause specifies a datafile named 'DISKC:DBONE.DAT', 30 megabytes in size.

All of these *filespecs* specify a value for the SIZE parameter and omit the REUSE clause, so none of these files can already exist. Oracle must create them.

The following statement adds another redo log file group with two members to the PAYABLE database:

```
ALTER DATABASE payable
   ADD LOGFILE GROUP 3 ('diska:log3.log', 'diskb:log3.log')
   SIZE 50K REUSE;
```

The *filespec* in the ADD LOGFILE clause specifies a new redo log file group with the GROUP value 3. This new group has members named 'DISKA:LOG3.LOG' and 'DISKB:LOG3.LOG', each 50 kilobytes in size. Because the *filespec* specifies the REUSE clause, each member can already exist. If a member exists, it must have a size of 50 kilobytes. If it does not exist, Oracle creates it with that size.

The following statement creates a tablespace named STOCKS that has three datafiles:

```
CREATE TABLESPACE stocks
   DATAFILE 'diskc:stock1.dat',
            'diskc:stock2.dat',
            'diskc:stock3.dat';
```

The *filespecs* for the datafiles specifies files named 'DISKC:STOCK1.DAT', 'DISKC:STOCK2.DAT', and 'DISKC:STOCK3.DAT'. Since each *filespec* omits the SIZE parameter, each file must already exist.

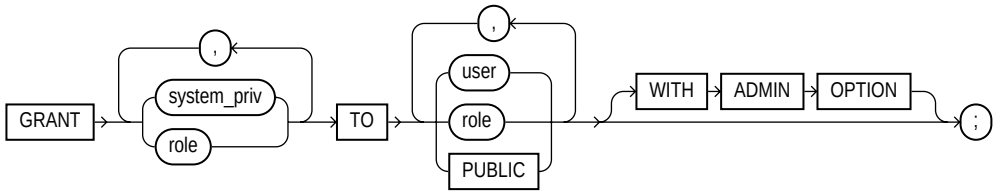
The following statement alters the STOCKS tablespace and adds a new datafile:

```
ALTER TABLESPACE stocks
   ADD DATAFILE 'diskc:stock4.dat' REUSE;
```

The *filespec* specifies a datafile named 'DISKC:STOCK4.DAT'. Since the *filespec* omits the SIZE parameter, the file must already exist and the REUSE clause is not significant.

GRANT *system_privileges_and_roles*

Syntax



Purpose

To grant system privileges and roles to users and roles. Both privileges and roles are either local, global, or external (see "CREATE USER" on page 7-425 and "CREATE ROLE" on page 7-344 for definitions).

You can authorize database users to use roles through means other than the database and the GRANT statement. For example, some operating systems have facilities that grant operating system privileges to operating system users. You can use such facilities to grant roles to Oracle users with the initialization parameter OS_ROLES. If you choose to grant roles to users through operating system facilities, you cannot also grant roles to users with the GRANT statement, although you can use the GRANT statement to grant system privileges to users and system privileges and roles to other roles. For information about other authorization methods, see *Oracle8i Administrator's Guide*.

For information on granting object privileges, see "GRANT object_privileges" on page 7-505.

Prerequisites

To grant a system privilege, you must either have been granted the system privilege with the ADMIN OPTION or have been granted the GRANT ANY PRIVILEGE system privilege.

To grant a role, you must either have been granted the role with the ADMIN OPTION or have been granted the GRANT ANY ROLE system privilege, or you must have created the role.

Keywords and Parameters

system_priv is a system privilege to be granted. Table 7–5 lists the system privileges (organized by the database object operated upon).

- If you grant **a privilege to a user**, Oracle adds the privilege to the user's privilege domain. The user can immediately exercise the privilege.
- If you grant **a privilege to a role**, Oracle adds the privilege to the role's privilege domain. Users who have been granted and have enabled the role can immediately exercise the privilege. Other users who have been granted the role can enable the role and exercise the privilege.
- If you grant **a privilege to PUBLIC**, Oracle adds the privilege to the privilege domains of each user. All users can immediately perform operations authorized by the privilege.

Restrictions:

- A privilege or role cannot appear more than once in the list of privileges and roles to be granted.
- You cannot grant a role to itself.
- You cannot grant a role IDENTIFIED GLOBALLY to anything.
- You cannot grant a role IDENTIFIED EXTERNALLY to a global user or global role.
- You cannot grant roles circularly. For example, if you grant the role BANKER to the role TELLER, you cannot subsequently grant TELLER to BANKER.

role is a role to be granted. You can grant an Oracle predefined role or a user-defined role. Table 7–6 lists the predefined roles. For information on creating a user-defined role, see "CREATE ROLE" on page 7-344.

- If you grant **a role to a user**, Oracle makes the role available to the user. The user can immediately enable the role and exercise the privileges in the role's privilege domain.
- If you grant **a role to another role**, Oracle adds the granted role's privilege domain to the grantee role's privilege domain. Users who have been granted the grantee role can enable it and exercise the privileges in the granted role's privilege domain.
- If you grant **a role to PUBLIC**, Oracle makes the role available to all users. All users can immediately enable the role and exercise the privileges in the roles privilege domain.

TO identifies users or roles to which system privileges and roles are granted.

Restriction: A user, role, or PUBLIC cannot appear more than once in the TO clause.

PUBLIC grants system privileges or roles to all users.

WITH ADMIN OPTION	enables the grantee to ■ Grant the role to another user or role, unless the role is a GLOBAL role ■ Revoke the role from another user or role ■ Alter the role to change the authorization needed to access it ■ Drop the role If you grant a system privilege or role to a user without specifying WITH ADMIN OPTION, and then subsequently grant the privilege or role to the user WITH ADMIN OPTION, the user has the ADMIN OPTION on the privilege or role. To revoke the admin option on a system privilege or role from a user, you must revoke the privilege or role from the user altogether and then grant the privilege or role to the user without the admin option.

Table 7–5 System Privileges

System Privilege	Allows grantee to . . .
CLUSTERS	
CREATE CLUSTER	Create clusters in grantee's schema
CREATE ANY CLUSTER	Create a cluster in any schema except SYS. Behaves similarly to CREATE ANY TABLE.
ALTER ANY CLUSTER	Alter clusters in any schema except SYS
DROP ANY CLUSTER	Drop clusters in any schema except SYS
CONTEXTS	
CREATE ANY CONTEXT	Create any context namespace
DROP ANY CONTEXT	Drop any context namespace
DATABASE	
ALTER DATABASE	Alter the database
ALTER SYSTEM	Issue ALTER SYSTEM statements
AUDIT SYSTEM	Issue AUDIT *sql_statements* statements
DATABASE LINKS	
CREATE DATABASE LINK	Create private database links in grantee's schema
CREATE PUBLIC DATABASE LINK	Create public database links

Table 7–5 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
DROP PUBLIC DATABASE LINK	Drop public database links
DIMENSIONS	
CREATE DIMENSION	Create dimensions in the grantee's schema
CREATE ANY DIMENSION	Create dimensions in any schema except SYS
ALTER ANY DIMENSION	Alter dimensions in any schema except SYS
DROP ANY DIMENSION	Drop dimensions in any schema except SYS
DIRECTORIES	
CREATE ANY DIRECTORY	Create directory database objects
DROP ANY DIRECTORY	Drop directory database objects
INDEXTYPES	
CREATE INDEXTYPE	Create an indextype in the grantee's schema
CREATE ANY INDEXTYPE	Create an indextype in any schema except SYS
DROP ANY INDEXTYPE	Drop an indextype in any schema except SYS
EXECUTE ANY INDEXTYPE	Reference an indextype in any schema except SYS
INDEXES	
CREATE INDEX	**C**reate in the grantee's schema an index on any table in the grantee's schema or a domain index
CREATE ANY INDEX	**C**reate in any schema except SYS a domain index or an index on any table in any schema except SYS
ALTER ANY INDEX	Alter indexes in any schema except SYS
DROP ANY INDEX	Drop indexes in any schema except SYS
QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables and views that are in the grantee's own schema.
GLOBAL QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables or views in any schema except SYS.

Table 7–5 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
LIBRARIES	
CREATE LIBRARY	Create external procedure / function libraries in grantee's schema
CREATE ANY LIBRARY	Create external procedure / function libraries in any schema except SYS
DROP LIBRARY	Drop external procedure / function libraries in the grantee's schema
DROP ANY LIBRARY	Drop external procedure / function libraries in any schema except SYS
MATERIALIZED VIEWS (which are identical to SNAPSHOTS)	
CREATE MATERIALIZED VIEW	Create a materialized view in the grantee's schema
CREATE ANY MATERIALIZED VIEW	Create materialized views in any schema except SYS
ALTER ANY MATERIALIZED VIEW	Alter materialized views in any schema except SYS
DROP ANY MATERIALIZED VIEW	Drop materialized views in any schema except SYS
QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables and views that are in the grantee's own schema.
GLOBAL QUERY REWRITE	Enable rewrite using a materialized view, or create a function-based index, when that materialized view or index references tables or views in any schema except SYS.
OPERATORS	
CREATE OPERATOR	Create an operator and its bindings in the grantee's schema
CREATE ANY OPERATOR	Create an operator and its bindings in any schema except SYS
DROP ANY OPERATOR	Drop an operator in any schema except SYS
EXECUTE ANY OPERATOR	Reference an operator in any schema except SYS
OUTLINES	

Table 7–5 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
CREATE ANY OUTLINE	Create outlines that can be used in any schema that uses outlines
ALTER ANY OUTLINE	Modify outlines.
DROP ANY OUTLINE	Drop outlines
PROCEDURES	
CREATE PROCEDURE	Create stored procedures, functions, and packages in grantee's schema
CREATE ANY PROCEDURE	Create stored procedures, functions, and packages in any schema except SYS
ALTER ANY PROCEDURE	Alter stored procedures, functions, or packages in any schema except SYS
DROP ANY PROCEDURE	Drop stored procedures, functions, or packages in any schema except SYS
EXECUTE ANY PROCEDURE	Execute procedures or functions (standalone or packaged)
	Reference public package variables in any schema except SYS
PROFILES	
CREATE PROFILE	Create profiles
ALTER PROFILE	Alter profiles
DROP PROFILE	Drop profiles
ROLES	
CREATE ROLE	Create roles
ALTER ANY ROLE	Alter any role in the database
DROP ANY ROLE	Drop roles
GRANT ANY ROLE	Grant any role in the database
ROLLBACK SEGMENTS	
CREATE ROLLBACK SEGMENT	Create rollback segments
ALTER ROLLBACK SEGMENT	Alter rollback segments
DROP ROLLBACK SEGMENT	Drop rollback segments

Table 7–5 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
SEQUENCES	
CREATE SEQUENCE	Create sequences in grantee's schema
CREATE ANY SEQUENCE	Create sequences in any schema except SYS
ALTER ANY SEQUENCE	Alter any sequence in the database
DROP ANY SEQUENCE	Drop sequences in any schema except SYS
SELECT ANY SEQUENCE	Reference sequences in any schema except SYS
SESSIONS	
CREATE SESSION	Connect to the database
ALTER RESOURCE COST	Set costs for session resources
ALTER SESSION	Issue ALTER SESSION statements
RESTRICTED SESSION	Logon after the instance is started using the SQL*Plus STARTUP RESTRICT statement
SNAPSHOTS (which are identical to MATERIALIZED VIEWS)	
CREATE SNAPSHOT	Create snapshots in grantee's schema
CREATE ANY SNAPSHOT	Create snapshots in any schema except SYS
ALTER ANY SNAPSHOT	Alter any snapshot in the database
DROP ANY SNAPSHOT	Drop snapshots in any schema except SYS
GLOBAL QUERY REWRITE	Enable rewrite using a snapshot, or create a function-based index, when that snapshot or index references tables or views in any schema except SYS.
QUERY REWRITE	Enable rewrite using a snapshot, or create a function-based index, when that snapshot or index references tables and views that are in the grantee's own schema.
SYNONYMS	
CREATE SYNONYM	Create synonyms in grantee's schema
CREATE ANY SYNONYM	Create private synonyms in any schema except SYS
CREATE PUBLIC SYNONYM	Create public synonyms
DROP ANY SYNONYM	Drop private synonyms in any schema except SYS

Table 7–5 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
DROP PUBLIC SYNONYM	Drop public synonyms
TABLES	
CREATE ANY TABLE	Create tables in any schema except SYS. The owner of the schema containing the table must have space quota on the tablespace to contain the table.
ALTER ANY TABLE	Alter any table or view in the schema
BACKUP ANY TABLE	Use the Export utility to incrementally export objects from the schema of other users
DELETE ANY TABLE	Delete rows from tables, table partitions, or views in any schema except SYS
DROP ANY TABLE	Drop or truncate tables or table partitions in any schema except SYS
INSERT ANY TABLE	Insert rows into tables and views in any schema except SYS
LOCK ANY TABLE	Lock tables and views in any schema except SYS
UPDATE ANY TABLE	Update rows in tables and views in any schema except SYS
SELECT ANY TABLE	Query tables, views, or snapshots in any schema except SYS
TABLESPACES	
CREATE TABLESPACE	Create tablespaces
ALTER TABLESPACE	Alter tablespaces
DROP TABLESPACE	Drop tablespaces
MANAGE TABLESPACE	Take tablespaces offline and online and begin and end tablespace backups
UNLIMITED TABLESPACE	Use an unlimited amount of any tablespace. This privilege overrides any specific quotas assigned. If you revoke this privilege from a user, the user's schema objects remain but further tablespace allocation is denied unless authorized by specific tablespace quotas. You cannot grant this system privilege to roles.
TRIGGERS	
CREATE TRIGGER	Create a database trigger in grantee's schema

Table 7–5 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
CREATE ANY TRIGGER	Create database triggers in any schema except SYS
ALTER ANY TRIGGER	Enable, disable, or compile database triggers in any schema except SYS
DROP ANY TRIGGER	Drop database triggers in any schema except SYS
ADMINISTER DATABASE TRIGGER	Create a trigger on DATABASE. (You must also have the CREATE TRIGGER or CREATE ANY TRIGGER privilege.)
TYPES	
CREATE TYPE	Create object types and object type bodies in grantee's schema
CREATE ANY TYPE	Create object types and object type bodies in any schema except SYS
ALTER ANY TYPE	Alter object types in any schema except SYS
DROP ANY TYPE	Drop object types and object type bodies in any schema except SYS
EXECUTE ANY TYPE	Use and reference object types and collection types in any schema except SYS, and invoke methods of an object type in any schema *if you make the grant to a specific user.* If you grant EXECUTE ANY TYPE to a role, users holding the enabled role will not be able to invoke methods of an object type in any schema.
USERS	
CREATE USER	Create users. This privilege also allows the creator to ■ assign quotas on **any** tablespace ■ set default and temporary tablespaces ■ assign a profile as part of a CREATE USER statement
ALTER USER	Alter any user. This privilege authorizes the grantee to ■ Change another user's password or authentication method, ■ Assign quotas on **any** tablespace, ■ Set default and temporary tablespaces, and ■ Assign a profile and default roles

Table 7–5 (Cont.) System Privileges

System Privilege	Allows grantee to . . .
BECOME USER	Become another user. (Required by any user performing a full database import.)
DROP USER	Drop users
VIEWS	
CREATE VIEW	Create views in grantee's schema
CREATE ANY VIEW	Create views in any schema except SYS
DROP ANY VIEW	Drop views in any schema except SYS
MISCELLANEOUS	
ANALYZE ANY	Analyze any table, cluster, or index in any schema except SYS
AUDIT ANY	Audit any object in any schema except SYS using AUDIT *schema_objects* statements
COMMENT ANY TABLE	Comment on any table, view, or column in any schema except SYS
FORCE ANY TRANSACTION	Force the commit or rollback of any in-doubt distributed transaction in the local database
	Induce the failure of a distributed transaction
FORCE TRANSACTION	Force the commit or rollback of grantee's in-doubt distributed transactions in the local database
GRANT ANY PRIVILEGE	Grant any system privilege.
SYSDBA	Perform STARTUP and SHUTDOWN operations
	ALTER DATABASE: open, mount, back up, or change character set
	CREATE DATABASE
	ARCHIVELOG and RECOVERY
	Includes the RESTRICTED SESSION privilege
SYSOPER	Perform STARTUP and SHUTDOWN operations
	ALTER DATABASE OPEN/MOUNT/BACKUP
	ARCHIVELOG and RECOVERY
	Includes the RESTRICTED SESSION privilege

Table 7–6 Oracle Predefined Roles

Predefined Role	Purpose
CONNECT, RESOURCE, and DBA	These roles are provided for compatibility with previous versions of Oracle. You should not rely on these roles, because they may not be created automatically by future versions of Oracle. Rather, Oracle recommends that you to design your own roles for database security.
DELETE_CATALOG_ROLE EXECUTE_CATALOG_ROLE SELECT_CATALOG_ROLE	These roles are provided for accessing exported data dictionary views and packages. For more information on these roles, see *Oracle8i Application Developer's Guide - Fundamentals.*
EXP_FULL_DATABASE IMP_FULL_DATABASE	These roles are provided for convenience in using the Import and Export utilities. For more information on these roles, see *Oracle8i Utilities.*
AQ_USER_ROLE AQ_ADMINISTRATOR_ROLE	You need these roles to use Oracle's Advanced Queuing functionality. For more information on these roles, see *Oracle8i Application Developer's Guide - Advanced Queuing.*
SNMPAGENT	This role is used by Enterprise Manager / Intelligent Agent. For more information, see *Oracle Enterprise Manager Administrator's Guide.*
RECOVERY_CATALOG_OWNER	You need this role to create a user who owns a recovery catalog. For more information on recovery catalogs, see *Oracle8i Backup and Recovery Guide.*
HS_ADMIN_ROLE	A DBA using Oracle's heterogeneous services feature needs this role to access appropriate tables in the data dictionary and to manipulate them with the DBMS_HS package. For more information, refer to *Oracle8i Distributed Database Systems* and *Oracle8i Supplied Packages Reference.*

Oracle also creates other roles that authorize you to administer the database. On many operating systems, these roles are called OSOPER and OSDBA. Their names may be different on your operating system.

Examples

To grant the CREATE SESSION system privilege to RICHARD, allowing RICHARD to log on to Oracle, issue the following statement:

```
GRANT CREATE SESSION
   TO richard;
```

To grant the CREATE TABLE system privilege to the role TRAVEL_AGENT, issue the following statement:

```
GRANT CREATE TABLE
   TO travel_agent;
```

TRAVEL_AGENT's privilege domain now contains the CREATE TABLE system privilege.

The following statement grants the TRAVEL_AGENT role to the EXECUTIVE role:

```
GRANT travel_agent
   TO executive;
```

TRAVEL_AGENT is now granted to EXECUTIVE. EXECUTIVE's privilege domain contains the CREATE TABLE system privilege.

To grant the EXECUTIVE role with the ADMIN OPTION to THOMAS, issue the following statement:

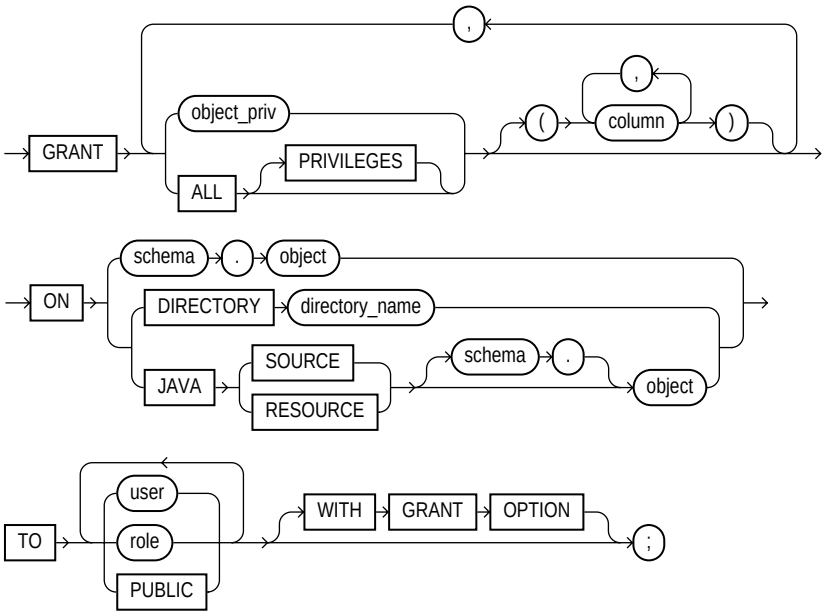
```
GRANT executive
   TO thomas
   WITH ADMIN OPTION;
```

THOMAS can now perform the following operations with the EXECUTIVE role:

- Enable the role and exercise any privileges in the role's privilege domain, including the CREATE TABLE system privilege
- Grant and revoke the role to and from other users
- Drop the role

GRANT *object_privileges*

Syntax



Purpose

To grant privileges for a particular object to users, roles, and PUBLIC. To grant system privileges and roles, use the GRANT *system_privileges_and_roles* statement described in the previous section of this chapter. Table 7–7 summarizes the object privileges that you can grant on each type of object.

If you grant **a privilege to a user,** Oracle adds the privilege to the user's privilege domain. The user can immediately exercise the privilege.

If you grant **a privilege to a role,** Oracle adds the privilege to the role's privilege domain. Users who have been granted and have enabled the role can immediately exercise the privilege. Other users who have been granted the role can enable the role and exercise the privilege.

If you grant **a privilege to PUBLIC,** Oracle adds the privilege to the privilege domain of each user. All users can immediately exercise the privilege.

Table 7–8 lists object privileges and the operations that they authorize. You can grant any of these system privileges with the GRANT statement.

For information on granting system privileges and roles, see "GRANT system_privileges_and_roles" on page 7-493. For information on revoking object grants, see "REVOKE schema_object_privileges" on page 7-532.

Prerequisites

You must own the object or the owner of the object must have granted you the object privileges with the GRANT OPTION. This rule applies to users with the DBA role.

Keywords and Parameters

object_priv	is an object privilege to be granted. You can substitute any of the values shown in Table 7–7. See also Table 7–8. **Restriction:** A privilege cannot appear more than once in the list of privileges to be granted.
ALL [PRIVILEGES]	grants all the privileges for the object that you have been granted with the GRANT OPTION. The user who owns the schema containing an object automatically has all privileges on the object with the GRANT OPTION. (The keyword PRIVILEGES is optional.)
column	specifies a table or view column on which privileges are granted. You can specify columns only when granting the INSERT, REFERENCES, or UPDATE privilege. If you do not list columns, the grantee has the specified privilege on all columns in the table or view.
ON	identifies the object on which the privileges are granted. Directory schema objects and Java source and resource schema objects are identified separately because they reside in separate namespaces.

	object	identifies the schema object on which the privileges are granted. If you do not qualify object with *schema*, Oracle assumes the object is in your own schema. The object can be one of the following types (see Table 7–7): ■ table, view, or materialized view / snapshot ■ sequence ■ procedure, function, or package ■ user-defined type ■ synonym for any of the above items ■ directory, library, operator, or indextype ■ a Java source, class, or resource **Note:** You cannot grant privileges directly to a single partition of a partitioned table. For information on how to grant privileges to a single partition indirectly, refer to *Oracle8i Concepts*.
	DIRECTORY	identifies a directory schema object on which privileges are granted by the DBA. You cannot qualify *directory_name* with a schema name. See "CREATE DIRECTORY" on page 7-264.
	JAVA SOURCE | RESOURCE	identifies a Java source or resource schema object on which privileges are granted. See "CREATE JAVA" on page 7-293.
TO	identifies users or roles to which the object privilege is granted. **Restriction:** A user or role cannot appear more than once in the TO clause.	
	PUBLIC	grants object privileges to all users.
WITH GRANT OPTION	allows the grantee to grant the object privileges to other users and roles. The grantee must be a user or PUBLIC, rather than a role.	

Table 7–7 Object Privileges

Object Privilege	Table	View	Sequence	Procedures, Functions, Packages[a]	Materialized View / Snapshot	Directory	Library	User-defined Type	Operator	Indextype
ALTER	X		X							
DELETE	X	X			X[b]					
EXECUTE				X			X	X	X	X
INDEX	X									
INSERT	X	X			X[b]					
READ						X				
REFERENCES	X									
SELECT	X	X	X		X					
UPDATE	X	X			X[b]					

[a]Oracle treats a Java class, source, or resource as if it were a procedure for purposes of granting object privileges.
[b]The DELETE, INSERT, and UPDATE privileges can be granted only to *updatable* materialized views.

Table 7–8 Object Privileges and the Operations They Authorize

Object Privilege	Allows Grantee to . . .
The following **table privileges** authorize operations on a table. Any one of following object privileges allows the grantee to lock the table in any lock mode with the LOCK TABLE statement.	
ALTER	Change the table definition with the ALTER TABLE statement.
DELETE	Remove rows from the table with the DELETE statement. **Note:** You must grant the SELECT privilege on the table along with the DELETE privilege.
INDEX	Create an index on the table with the CREATE INDEX statement.
INSERT	Add new rows to the table with the INSERT statement.
REFERENCES	Create a constraint that refers to the table. You cannot grant this privilege to a role.

Table 7–8 (Cont.) Object Privileges and the Operations They Authorize

Object Privilege	Allows Grantee to . . .
SELECT	Query the table with the SELECT statement.
UPDATE	Change data in the table with the UPDATE statement. **Note:** You must grant the SELECT privilege on the table along with the UPDATE privilege.
The following **view privileges** authorize operations on a view. Any one of the following object privileges allows the grantee to lock the view in any lock mode with the LOCK TABLE statement. To grant a privilege on a view, you must have that privilege with the GRANT OPTION on all of the view's base tables.	
DELETE	Remove rows from the view with the DELETE statement.
INSERT	Add new rows to the view with the INSERT statement.
SELECT	Query the view with the SELECT statement.
UPDATE	Change data in the view with the UPDATE statement.
The following **sequence privileges** authorize operations on a sequence.	
ALTER	Change the sequence definition with the ALTER SEQUENCE statement.
SELECT	Examine and increment values of the sequence with the CURRVAL and NEXTVAL pseudocolumns.
The following **procedure, function, and package privilege** authorizes operations on procedures, functions, or packages. This privilege also applies to **Java sources, classes, and resources**, which Oracle treats as though they were procedures for purposes of granting object privileges.	
EXECUTE	Compile the procedure or function or execute it directly, or access any program object declared in the specification of a package. **Note:** Users do not need this privilege to execute a procedure, function, or package indirectly. For more information, refer to *Oracle8i Concepts* and *Oracle8i Application Developer's Guide - Fundamentals.*
The following **snapshot privilege** authorizes operations on a snapshot.	
SELECT	Query the snapshot with the SELECT statement.

Table 7–8 (Cont.) Object Privileges and the Operations They Authorize

Object Privilege	Allows Grantee to . . .
Synonym privileges are the same as the privileges for the base object. Granting a privilege on a synonym is equivalent to granting the privilege on the base object. Similarly, granting a privilege on a base object is equivalent to granting the privilege on all synonyms for the object. If you grant a user a privilege on a synonym, the user can use either the synonym name or the base object name in the SQL statement that exercises the privilege.	
The following **directory privilege** provides secured access to the files stored in the operating system directory to which the directory object serves as a pointer. The directory object contains the full pathname of the operating system directory where the files reside. Because the files are actually stored outside the database, Oracle server processes also need to have appropriate file permissions on the file system server. Granting object privileges on the directory database object to individual database users, rather than on the operating system, allows Oracle to enforce security during file operations.	
READ	Read files in the directory.
The following **object type privilege** authorizes operations on an object type	
EXECUTE	Use and reference the specified object and to invoke its methods.
The following **indextype privilege** authorizes operations on indextypes.	
EXECUTE	Reference an indextype.
The following **operator privilege** authorizes operations on user-defined operators.	
EXECUTE	Reference an operator.

Examples

To grant READ on directory BFILE_DIR1 to user SCOTT, with the GRANT OPTION, issue the following statement:

```
GRANT READ ON DIRECTORY bfile_dir1 TO scott
   WITH GRANT OPTION;
```

To grant all privileges on the table BONUS to the user JONES with the GRANT OPTION, issue the following statement:

```
GRANT ALL ON bonus TO jones
   WITH GRANT OPTION;
```

JONES can subsequently perform the following operations:

- Exercise any privilege on the BONUS table
- Grant any privilege on the BONUS table to another user or role

To grant SELECT and UPDATE privileges on the view GOLF_HANDICAP to all users, issue the following statement:

```
GRANT SELECT, UPDATE
   ON golf_handicap TO PUBLIC;
```

All users can subsequently query and update the view of golf handicaps.

To grant SELECT privilege on the ESEQ sequence in the schema ELLY to the user BLAKE, issue the following statement:

```
GRANT SELECT
   ON elly.eseq TO blake;
```

BLAKE can subsequently generate the next value of the sequence with the following statement:

```
SELECT elly.eseq.NEXTVAL
   FROM DUAL;
```

To grant BLAKE the REFERENCES privilege on the EMPNO column and the UPDATE privilege on the EMPNO, SAL, and COMM columns of the EMP table in the schema SCOTT, issue the following statement:

```
GRANT REFERENCES (empno), UPDATE (empno, sal, comm)
   ON scott.emp
   TO blake;
```

BLAKE can subsequently update values of the EMPNO, SAL, and COMM columns. BLAKE can also define referential integrity constraints that refer to the EMPNO column. However, because the GRANT statement lists only these columns, BLAKE cannot perform operations on any of the other columns of the EMP table.

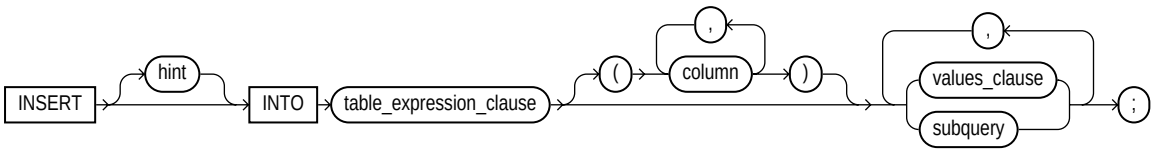
For example, BLAKE can create a table with a constraint:

```
CREATE TABLE dependent
   (dependno   NUMBER,
    dependname VARCHAR2(10),
    employee   NUMBER
   CONSTRAINT in_emp REFERENCES scott.emp(empno) );
```

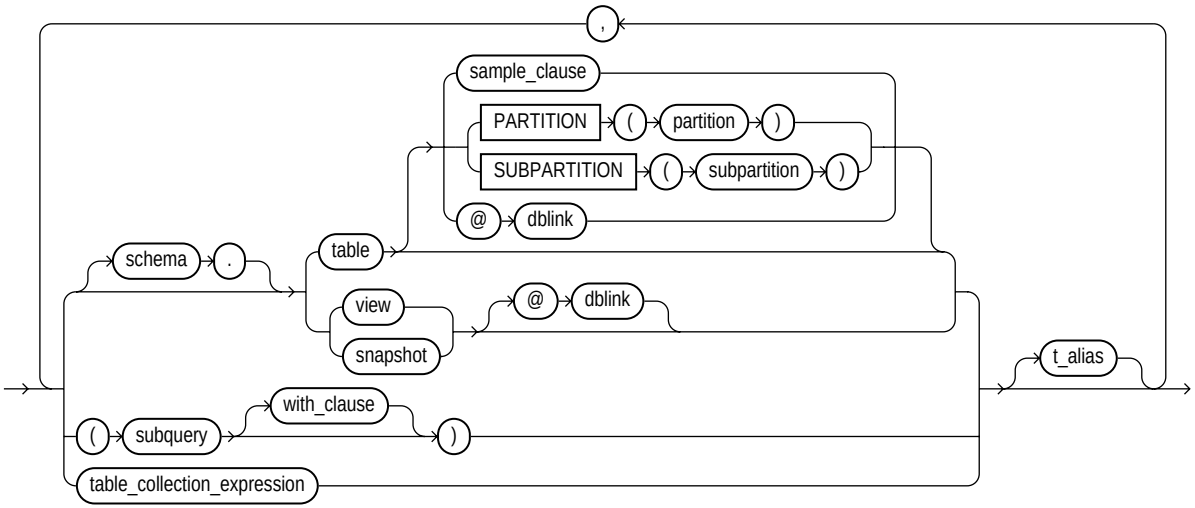
The constraint IN_EMP ensures that all dependents in the DEPENDENT table correspond to an employee in the EMP table in the schema SCOTT.

INSERT

Syntax

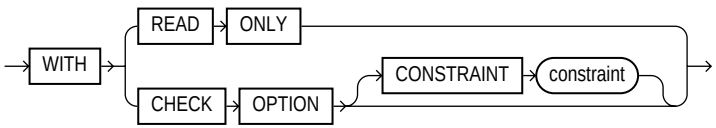


table_expression_clause::=



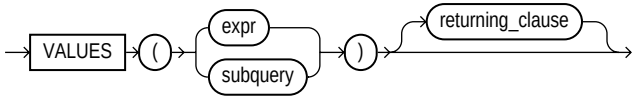
subquery: see "SELECT and Subqueries" on page 7-541.

with_clause::=

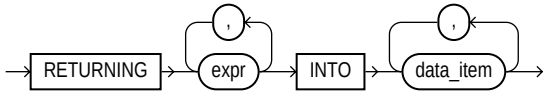


table_collection_expression::=

values_clause::=



returning_clause::=



Purpose

To add rows to a table, a view's base table, a partition of a partitioned table or a subpartition of a composite-partitioned table, or an object table or an object view's base table.

Prerequisites

For you to insert rows into a table, the table must be in your own schema or you must have INSERT privilege on the table.

For you to insert rows into the base table of a view, the owner of the schema containing the view must have INSERT privilege on the base table. Also, if the view is in a schema other than your own, you must have INSERT privilege on the view.

If you have the INSERT ANY TABLE system privilege, you can also insert rows into any table or any view's base table.

Keywords and Parameters

hint is a comment that passes instructions to the optimizer on choosing an execution plan for the statement. For the syntax and description of hints, see "Hints" on page 2-58 and *Oracle8i Tuning*.

table_expression_clause

schema is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table | *view* | *subquery* is the name of the table or object table, or view or object view, or the column or columns returned by a subquery, into which rows are to be inserted. If you specify a view or object view, Oracle inserts rows into the view's base table.

If any value to be inserted is a REF to an object table, and if the object table has a primary key object identifier, then the column into which you insert the REF must be a REF column with a referential integrity or SCOPE constraint to the object table.

If *table* (or the base table of *view*) contains one or more domain index columns, this statement executes the appropriate indextype insert routine. For more information on these routines, see *Oracle8i Data Cartridge Developer's Guide*.

Issuing an INSERT statement against a table fires any INSERT triggers defined on the table.

Restrictions:

- You cannot execute this statement if *table* (or the base table of *view*) contains any domain indexes marked LOADING or FAILED.
- You cannot specify the *sample_clause* in an INSERT statement.
- You cannot specify the ORDER BY clause in the subquery of the *table_expression_clause*.
- If a view was created using the WITH CHECK OPTION, then you can insert into the view only rows that satisfy the view's defining query.
- If a view was created using a single base table, then you can insert rows into the view and then retrieve those values using the *returning_clause*.

- You cannot insert rows into a view except with INSTEAD OF triggers if the view's defining query contains one of the following constructs:
  - A set operator
  - A DISTINCT operator
  - An aggregate function
  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause
  - A collection expression in a SELECT list
  - A subquery in a SELECT list
  - Joins (with some exceptions). See *Oracle8i Administrator's Guide* for details.
- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, the INSERT statement will fail unless the SKIP_UNUSABLE_INDEXES parameter has been set to TRUE. For more information, see "ALTER SESSION" on page 7-78.

PARTITION *(partition_name)* | SUBPARTITION *(subpartition_name)*

specifies the name of the partition or subpartition within *table* (or the base table of *view*) targeted for inserts.

If a row to be inserted does not map into a specified partition or subpartition, Oracle returns an error.

Restriction: This clause is not valid for object tables or object views.

dblink

is a complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see "Referring to Schema Objects and Parts" on page 2-71. You can insert rows into a remote table or view only if you are using Oracle's distributed functionality.

If you omit *dblink*, Oracle assumes that the table or view is on the local database.

with_clause

restricts the subquery in one of the following ways:

- WITH READ ONLY specifies that the subquery cannot be updated.
- WITH CHECK OPTION specifies that Oracle prohibits any changes to that table that would produce rows that are not included in the subquery. See the "WITH CHECK OPTION Example" on page 7-558.

table_collection_expression

informs Oracle that the collection value expression should be treated as a table. See "Table Collection Examples" on page 7-556.

collection_expression is a subquery that selects a nested table column from *table* or *view*.

Note: In earlier releases of Oracle, *table_collection_expression* was expressed as "THE subquery". That usage is now deprecated.

column

is a column of the table or view. In the inserted row, each column in this list is assigned a value from the *values_clause* or the subquery.

If you omit one of the table's columns from this list, the column's value for the inserted row is the column's default value as specified when the table was created. For more information on default column values, see "CREATE TABLE" on page 7-359. If any of these columns has a NOT NULL constraint, then Oracle returns an error indicating that the constraint has been violated and rolls back the INSERT statement.

If you omit the column list altogether, the *values_clause* or query must specify values for all columns in the table.

values_clause specifies a row of values to be inserted into the table or view. See the syntax description in "Expressions" on page 5-1 and "SELECT and Subqueries" on page 7-541. You must specify a value in the *values_clause* for each column in the column list. If you omit the column list, then the *values_clause* must provide values for every column in the table.

Restrictions:

- You cannot initialize an internal LOB attribute in an object with a value other than empty or null. That is, you cannot use a literal.
- You cannot insert a BFILE value until you have initialized the BFILE locator to null or to a directory alias and filename. See the "BFILE Example" on page 7-519. For information on initializing BFILEs, see *Oracle Call Interface Programmer's Guide* and *Oracle8i Application Developer's Guide - Fundamentals*.

Note: If you insert string literals into a RAW column, during subsequent queries, Oracle will perform a full table scan rather than using any index that might exist on the RAW column.

subquery is a subquery that returns rows that are inserted into the table. If the subquery selects no rows, Oracle inserts no rows into the table.

- When specified with VALUES, the subquery returns values to be inserted into one row.
- When specified without VALUES, the subquery can return values to be inserted into more than one row.

The subquery can refer to any table, view, or snapshot, including the target table of the INSERT statement. The select list of this subquery must have the same number of columns as the column list of the INSERT statement. If you omit the column list, then the subquery must provide values for every column in the table. See "SELECT and Subqueries" on page 7-541.

You can use *subquery* in combination with the TO_LOB function to convert the values in a LONG column to LOB values in another column in the same or another table. For a discussion of why and when to copy LONGs to LOBs, see *Oracle8i Migration*. For a description of how to use the TO_LOB function, see "Conversion Functions" on page 4-4. See also the TO_LOB Example on page 7-519. To migrate LONGs to LOBs in a view, you must perform the migration on the base table, and then add the LOB to the view.

Note: If *subquery* returns (in part or totally) the equivalent of an existing materialized view, Oracle may use the materialized view (for query rewrite) in place of one or more tables specified in *subquery*. For more information on materialized views and query rewrite, see *Oracle8i Tuning*.

t_alias provides a **correlation name** for the table, view, or subquery to be referenced elsewhere in the statement.

returning_clause retrieves the rows affected by the INSERT.

An INSERT statement with a *returning_clause* retrieves the rows inserted and stores them in PL/SQL variables or bind variables. Using a *returning_clause* in INSERT statements with a *values_clause* enables you to return column expressions, ROWIDs, and REFs and store them in output bind variables. You can also use INSERT with a *returning_clause* for views with single base tables.

expr is some form of the syntax descriptions in "Expressions" on page 5-1. You must specify a column expression in the *returning_clause* for each variable in the *data_item_list*.

INTO indicates that the values of the changed rows are to be stored in the variable(s) specified in *data_item_list*.

data_item is a PL/SQL variable or bind variable that stores a retrieved *expr* value.

Restrictions:

- You cannot use this clause with Parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.

Examples

VALUES Examples The following statement inserts a row into the DEPT table:

```
INSERT INTO dept
   VALUES (50, 'PRODUCTION', 'SAN FRANCISCO');
```

The following statement inserts a row with six columns into the EMP table. One of these columns is assigned NULL and another is assigned a number in scientific notation:

```
INSERT INTO emp (empno, ename, job, sal, comm, deptno)
   VALUES (7890, 'JINKS', 'CLERK', 1.2E3, NULL, 40);
```

The following statement has the same effect as the preceding example, but uses a subquery in the *table_expression_clause*:

```
INSERT INTO (SELECT empno, ename, job, sal, comm, deptno FROM emp)
   VALUES (7890, 'JINKS', 'CLERK', 1.2E3, NULL, 40);
```

Subquery Example The following statement copies managers and presidents or employees whose commission exceeds 25% of their salary into the BONUS table:

```
INSERT INTO bonus
   SELECT ename, job, sal, comm
   FROM emp
   WHERE comm > 0.25 * sal
   OR job IN ('PRESIDENT', 'MANAGER');
```

Database Link Example The following statement inserts a row into the ACCOUNTS table owned by the user SCOTT on the database accessible by the database link SALES:

```
INSERT INTO scott.accounts@sales (acc_no, acc_name)
   VALUES (5001, 'BOWER');
```

Assuming that the ACCOUNTS table has a BALANCE column, the newly inserted row is assigned the default value for this column (if one has been defined), because this INSERT statement does not specify a BALANCE value.

Sequence Example The following statement inserts a new row containing the next value of the employee sequence into the EMP table:

```
INSERT INTO emp
   VALUES  (empseq.nextval, 'LEWIS', 'CLERK',
            7902, SYSDATE, 1200, NULL, 20);
```

Partition Example The following example adds rows from LATEST_DATA into partition OCT98 of the SALES table:

```
INSERT INTO sales PARTITION (oct98)
   SELECT * FROM latest_data;
```

Bind Variable Example The following example returns the values of the inserted rows into output bind variables :BND1 and :BND2:

```
INSERT INTO emp VALUES (empseq.nextval, 'LEWIS', 'CLARK',
                        7902, SYSDATE, 1200, NULL, 20)
   RETURNING sal*12, job INTO :bnd1, :bnd2;
```

Bind Array Example The following example returns the reference value for the inserted row into bind array :1:

```
INSERT INTO employee
   VALUES ('Kitty Mine', 'Peaches Fuzz', 'Meena Katz')
   RETURNING REF(employee) INTO :1;
```

TO_LOB Example The following example copies LONG data to a LOB column in the following existing table:

```
CREATE TABLE long_tab (long_pics LONG RAW);
```

First you must create a table with a LOB.

```
CREATE TABLE lob_tab (lob_pics BLOB);
```

Next, use an INSERT ... SELECT statement to copy the data in all rows for the LONG column into the newly created LOB column:

```
INSERT INTO lob_tab (lob_pics)
   SELECT TO_LOB(long_pics) FROM long_tab;
```

Once you are confident that the migration has been successful, you can drop the LONG_PICS table. Alternatively, if the table contains other columns, you can simply drop the LONG column from the table as follows:

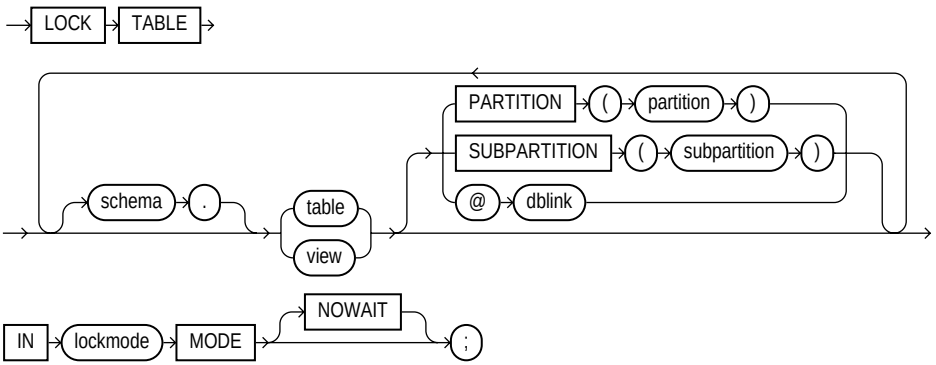
```
ALTER TABLE long_tab DROP COLUMN long_pics;
```

BFILE Example When you INSERT or UPDATE a BFILE, you must initialize it to null or to a directory alias and filename, as shown in the next example. Assume that the EMP table has a NUMBER column followed by a BFILE column:

```
INSERT INTO emp
   VALUES (1, BFILENAME ('a_dir_alias', 'a_filename'));
```

LOCK TABLE

Syntax



Purpose

To lock one or more tables (or table partitions or subpartitions) in a specified mode. This lock manually overrides automatic locking and permits or denies access to a table or view by other users for the duration of your operation.

Some forms of locks can be placed on the same table at the same time. Other locks allow only one lock per table. For a complete description of the interaction of lock modes, see *Oracle8i Concepts*.

A locked table remains locked until you either commit your transaction or roll it back, either entirely or to a savepoint before you locked the table. For more information, see "COMMIT" on page 7-214, "ROLLBACK" on page 7-537, and "SAVEPOINT" on page 7-539.

A lock never prevents other users from querying the table. A query never places a lock on a table. Readers never block writers and writers never block readers.

Prerequisites

The table or view must be in your own schema or you must have the LOCK ANY TABLE system privilege, or you must have any object privilege on the table or view.

Keywords and Parameters

schema is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table / view is the name of the table to be locked. If you specify *view*, Oracle locks the view's base tables.

If you specify PARTITION (*partition*) or SUBPARTITION (*subpartition*), Oracle first acquires an implicit lock on the table. The table lock is the same as the lock you specify for *partition* or *subpartition*, with two exceptions:

- If you specify a SHARE lock for the subpartition, Oracle acquires an implicit ROW SHARE lock on the table.
- If you specify an EXCLUSIVE lock for the subpartition, Oracle acquires an implicit ROW EXCLUSIVE lock on the table.

If you specify PARTITION and *table* is composite-partitioned, then Oracle acquires locks on all the subpartitions of *partition*.

dblink is a database link to a remote Oracle database where the table or view is located. For information on specifying database links, see "Referring to Objects in Remote Databases" on page 2-74. You can lock tables and views on a remote database only if you are using Oracle's distributed functionality. All tables locked by a LOCK TABLE statement must be on the same database.

If you omit *dblink*, Oracle assumes the table or view is on the local database.

lockmode is one of the following:

ROW SHARE allows concurrent access to the locked table, but prohibits users from locking the entire table for exclusive access. ROW SHARE is synonymous with SHARE UPDATE, which is included for compatibility with earlier versions of Oracle.

ROW EXCLUSIVE is the same as ROW SHARE, but also prohibits locking in SHARE mode. Row Exclusive locks are automatically obtained when updating, inserting, or deleting.

SHARE UPDATE—see ROW SHARE.

SHARE allows concurrent queries but prohibits updates to the locked table.

SHARE ROW EXCLUSIVE is used to look at a whole table and to allow others to look at rows in the table but to prohibit others from locking the table in SHARE mode or updating rows.

EXCLUSIVE allows queries on the locked table but prohibits any other activity on it.

NOWAIT specifies that Oracle returns control to you immediately if the specified table (or specified partition or subpartition) is already locked by another user. In this case, Oracle returns a message indicating that the table, partition, or subpartition is already locked by another user.

If you omit this clause, Oracle waits until the table is available, locks it, and returns control to you.

Examples

The following statement locks the EMP table in exclusive mode, but does not wait if another user already has locked the table:

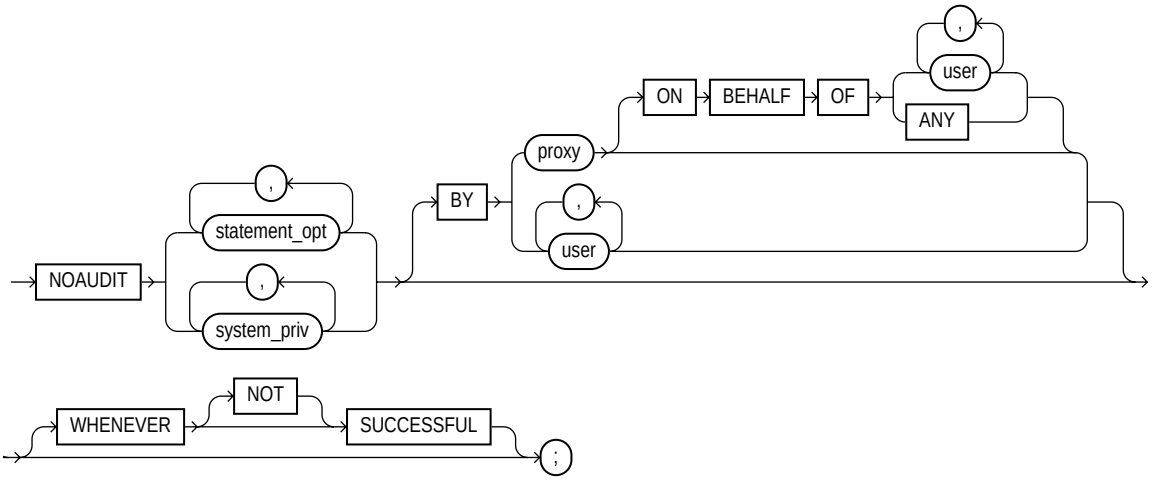
```
LOCK TABLE emp
    IN EXCLUSIVE MODE
    NOWAIT;
```

The following statement locks the remote ACCOUNTS table that is accessible through the database link BOSTON:

```
LOCK TABLE accounts@boston
    IN SHARE MODE;
```

NOAUDIT *sql_statements*

Syntax



Purpose

To stop auditing previously enabled by the AUDIT *sql_statements* statement. To stop auditing enabled by the AUDIT *schema_objects* statement, refer to "NOAUDIT schema_objects" on page 7-525.

The NOAUDIT statement must have the same syntax as the previous AUDIT statement. Further, it reverses the effects only of that particular statement. Therefore, if one AUDIT statement (statement A) enables auditing for a specific user, and a second (statement B) enables auditing for all users, then a NOAUDIT statement to disable auditing for all users (statement C) reverses statement B, but leaves statement A in effect and continues to audit the user that statement A specified. For information on auditing specific SQL statements, see the "AUDIT sql_statements" on page 7-197.

Prerequisites

You must have the AUDIT SYSTEM system privilege.

Keywords and Parameters

statement_opt	is a statement option for which auditing is stopped. For a list of the statement options and the SQL statements they audit, see Table 7–1 on page 7-200 and Table 7–2 on page 7-202.
system_priv	is a system privilege for which auditing is stopped. For a list of the system privileges and the statements they authorize, see Table 7–5 on page 7-495.
BY *user*	stops auditing only for SQL statements issued by specified users in their subsequent sessions. If you omit this clause, Oracle stops auditing for all users' statements, except for the situation described for WHENEVER SUCCESSFUL.
BY *proxy*	stops auditing only for the SQL statements issued by the specified proxy, on behalf of a specific user or any user.
WHENEVER SUCCESSFUL	stops auditing only for SQL statements that complete successfully. NOT stops auditing only for statements that result in Oracle errors. If you omit the WHENEVER SUCCESSFUL clause entirely, Oracle stops auditing for all statements, regardless of success or failure.

Examples

The following examples correspond to three examples listed in "AUDIT sql_ statements".

If you have chosen auditing for every SQL statement that creates or drops a role, you can stop auditing of such statements by issuing the following statement:

```
NOAUDIT ROLE;
```

If you have chosen auditing for any statement that queries or updates any table issued by the users SCOTT and BLAKE, you can stop auditing for SCOTT's queries by issuing the following statement:

```
NOAUDIT SELECT TABLE
   BY scott;
```

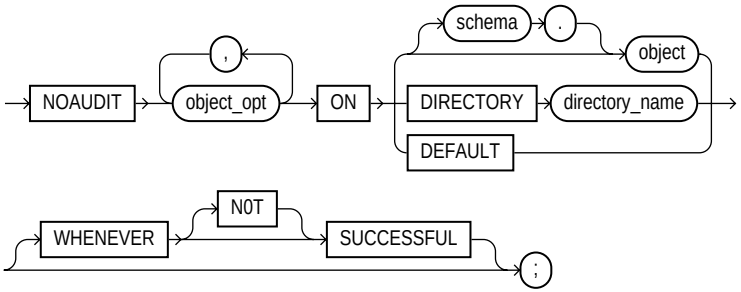
The above statement stops auditing only SCOTT's queries, so Oracle continues to audit BLAKE's queries and updates as well as SCOTT's updates.

To stop auditing on all statements that are authorized by DELETE ANY TABLE system privilege, issue the following statement:

```
NOAUDIT DELETE ANY TABLE;
```

NOAUDIT *schema_objects*

Syntax



Purpose

To stop auditing previously enabled by the AUDIT *schema_objects* statement. For more information on auditing, see "AUDIT schema_objects" on page 7-205.

To stop auditing enabled by the AUDIT *sql_statements* statement, refer to "NOAUDIT sql_statements" on page 7-523.

Prerequisites

The object on which you stop auditing must be in your own schema or you must have the AUDIT ANY system privilege. In addition, if the object you chose for auditing is a directory, even if you created it, you must have the AUDIT ANY system privilege.

Keywords and Parameters

object_opt	stops auditing for particular operations on the object. For a list of these options, see Table 7–3 on page 7-207.
ON	identifies the object on which auditing is stopped. If you do not qualify object with *schema*, Oracle assumes the object is in your own schema.
object	must a table, view, sequence, stored procedure, function, or package, snapshot, or library. For information on auditing specific schema objects, refer to "AUDIT schema_objects" on page 7-205.

DIRECTORY *directory_name*	identifies the name of the directory on which auditing is being stopped.
DEFAULT	removes the specified object options as default object options for subsequently created objects.
WHENEVER SUCCESSFUL	stops auditing only for SQL statements that complete successfully. NOT stops auditing only for statements that result in Oracle errors. If you omit the WHENEVER SUCCESSFUL clause entirely, Oracle stops auditing for all statements, regardless of success or failure.

Examples

If you have chosen auditing for every SQL statement that queries the EMP table in the schema SCOTT, you can stop auditing for such queries by issuing the following statement:

```
NOAUDIT SELECT
   ON scott.emp;
```

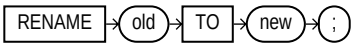
You can stop auditing for queries that complete successfully by issuing the following statement:

```
NOAUDIT SELECT
   ON scott.emp
   WHENEVER SUCCESSFUL;
```

This statement stops auditing only for successful queries. Oracle continues to audit queries resulting in Oracle errors.

RENAME

Syntax



Purpose

To rename a table, view, sequence, or private synonym for a table, view, or sequence.

- Oracle automatically transfers integrity constraints, indexes, and grants on the old object to the new object.
- Oracle invalidates all objects that depend on the renamed object, such as views, synonyms, and stored procedures and functions that refer to a renamed table.

Do not use this statement to rename public synonyms. Instead, drop the public synonym and then create another public synonym with the new name. See "DROP SYNONYM" on page 7-474 and "CREATE SYNONYM" on page 7-356.

Prerequisites

The object must be in your own schema.

Keywords and Parameters

old	is the name of an existing table, view, sequence, or private synonym.
new	is the new name to be given to the existing object. The new name must not already be used by another schema object in the same namespace and must follow the rules for naming schema objects defined in the section "Schema Object Naming Rules" on page 2-67.

Example

To change the name of table DEPT to EMP_DEPT, issue the following statement:

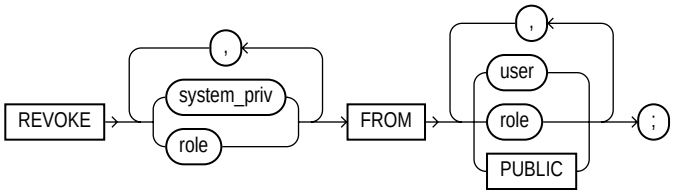
```
RENAME dept TO emp_dept;
```

You cannot use this statement directly to rename columns. However, you can rename a column using this statement together with the CREATE TABLE statement with AS *subquery*. The following statements re-create the table STATIC, renaming a column from OLDNAME to NEWNAME:

```
CREATE TABLE temporary (newname, col2, col3)
    AS SELECT oldname, col2, col3 FROM static;

DROP TABLE static;

RENAME temporary TO static;
```

REVOKE *system_privileges_and_roles*

Syntax



Purpose

To revoke system privileges and roles from users and roles. To revoke object privileges from users and roles, refer to "REVOKE schema_object_privileges" on page 7-532. For information on granting system privileges and roles, see "GRANT system_privileges_and_roles" on page 7-493.

Prerequisites

You must have been granted the system privilege or role with the ADMIN OPTION. Also, you can revoke any role if you have the GRANT ANY ROLE system privilege.

The REVOKE statement can revoke only privileges and roles that were previously granted directly with a GRANT statement. You cannot use this statement to revoke:

- Privileges or roles not granted to the revokee
- Roles granted through the operating system
- Privileges or roles granted to the revokee through roles

Keywords and Parameters

system_priv is a system privilege to be revoked. For a list of the system privileges, see Table 7–5 on page 7-495.

- If you revoke **a privilege from a user**, Oracle removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

- If you revoke **a privilege from a role**, Oracle removes the privilege from the role's privilege domain. Effective immediately, users with the role enabled cannot exercise the privilege. Also, other users who have been granted the role and subsequently enable the role cannot exercise the privilege.
- If you revoke **a privilege from PUBLIC,** Oracle removes the privilege from the privilege domain of each user who has been granted the privilege through PUBLIC. Effective immediately, such users can no longer exercise the privilege. However, the privilege is not revoked from users who have been granted the privilege directly or through roles.

Restriction: A system privilege cannot appear more than once in the list of privileges to be revoked.

role is a role to be revoked. For a list of the roles predefined by Oracle, see "GRANT system_privileges_and_roles" on page 7-493.

- If you revoke **a role from a user,** Oracle makes the role unavailable to the user. If the role is currently enabled for the user, the user can continue to exercise the privileges in the role's privilege domain as long as it remains enabled. However, the user cannot subsequently enable the role.
- If you revoke **a role from another role,** Oracle removes the revoked role's privilege domain from the revokee role's privilege domain. Users who have been granted and have enabled the revokee role can continue to exercise the privileges in the revoked role's privilege domain as long as the revokee role remains enabled. However, other users who have been granted the revokee role and subsequently enable it cannot exercise the privileges in the privilege domain of the revoked role.
- If you revoke **a role from PUBLIC,** Oracle makes the role unavailable to all users who have been granted the role through PUBLIC. Any user who has enabled the role can continue to exercise the privileges in its privilege domain as long as it remains enabled. However, users cannot subsequently enable the role. The role is not revoked from users who have been granted the role directly or through other roles.

Restriction: A system role cannot appear more than once in the list of roles to be revoked.

FROM identifies users and roles from which the system privileges or roles are to be revoked.

Restriction: A user, a role, or PUBLIC cannot appear more than once in the FROM clause.

PUBLIC revokes the system privilege or role from all users.

Examples

The following statement revokes the DROP ANY TABLE system privilege from the users BILL and MARY:

```
REVOKE DROP ANY TABLE
    FROM bill, mary;
```

BILL and MARY can no longer drop tables in schemas other than their own.

The following statement revokes the role CONTROLLER from the user HANSON:

```
REVOKE controller
    FROM hanson;
```

HANSON can no longer enable the CONTROLLER role.

The following statement revokes the CREATE TABLESPACE system privilege from the CONTROLLER role:

```
REVOKE CREATE TABLESPACE
   FROM controller;
```

Enabling the CONTROLLER role no longer allows users to create tablespaces.

To revoke the role VP from the role CEO, issue the following statement:

```
REVOKE vp
  FROM ceo;
```

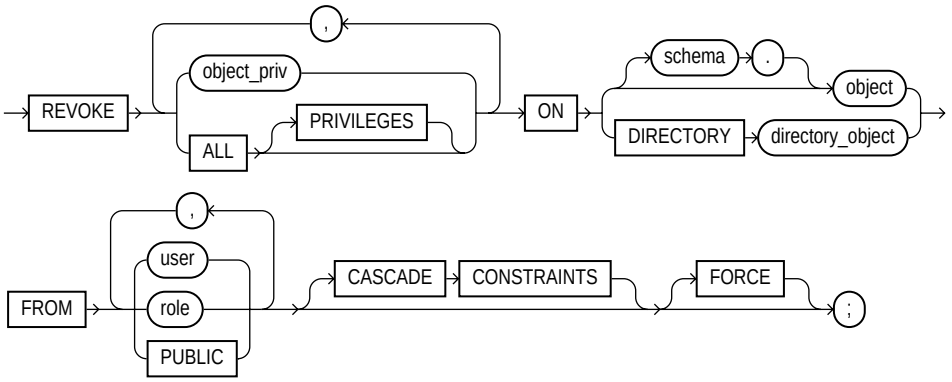
VP is no longer granted to CEO.

To revoke the CREATE ANY DIRECTORY system privilege from user SCOTT, issue the following statement:

```
REVOKE CREATE ANY DIRECTORY FROM scott;
```

REVOKE *schema_object_privileges*

Syntax



Purpose

To revoke object privileges for a particular object from users and roles.

For information on granting schema object privileges, see "GRANT object_ privileges" on page 7-505. To revoke system privileges or roles, refer to "REVOKE system_privileges_and_roles" on page 7-529.

Each object privilege authorizes some operation on an object. By revoking an object privilege, you prevent the revokee from performing that operation. However, multiple users may grant the same object privilege to the same user, role, or PUBLIC. To remove the privilege from the grantee's privilege domain, all grantors must revoke the privilege. If even one grantor does not revoke the privilege, the grantee can still exercise the privilege by virtue of that grant.

For a summary of the object privileges for each type of object, see Table 7–7 on page 7-508.

Prerequisites

You must have previously granted the object privileges to each user and role.

You can use the REVOKE statement only to revoke object privileges that you previously granted directly to the revokee. You cannot use this statement to revoke:

- Object privileges that you did not grant to the revokee
- Privileges granted through the operating system
- Privileges granted to roles granted to the revokee

Keywords and Parameters

object_priv is an object privilege to be revoked. You can substitute any of the following values: ALTER, DELETE, EXECUTE, INDEX, INSERT, READ, REFERENCES, SELECT, UPDATE.

- If you revoke **a privilege from a user,** Oracle removes the privilege from the user's privilege domain. Effective immediately, the user cannot exercise the privilege.

  - If that user has granted that privilege to other users or roles, Oracle also revokes the privilege from those other users or roles.

  - If that user's schema contains a procedure, function, or package that contains SQL statements that exercise the privilege, the procedure, function, or package can no longer be executed.

  - If that user's schema contains a view on that object, Oracle invalidates the view.

  - If you revoke the REFERENCES privilege from a user who has exercised the privilege to define referential integrity constraints, you must specify the CASCADE CONSTRAINTS clause.

- If you revoke **a privilege from a role,** Oracle removes the privilege from the role's privilege domain. Effective immediately, users with the role enabled cannot exercise the privilege. Other users who have been granted the role cannot exercise the privilege after enabling the role.
- If you revoke **a privilege from PUBLIC,** Oracle removes the privilege from the privilege domain of each user who has been granted the privilege through PUBLIC. Effective immediately, all such users are restricted from exercising the privilege. However, the privilege is not revoked from users who have been granted the privilege directly or through roles.

Restriction: A privilege cannot appear more than once in the list of privileges to be revoked. A user, a role, or PUBLIC cannot appear more than once in the FROM clause.

ALL PRIVILEGES revokes all object privileges that you have granted to the revokee.

Note: If *no* privileges have been granted on the object, Oracle takes no action and does not return an error.

ON DIRECTORY *directory_object* identifies a directory object on which privileges are revoked. You cannot qualify *directory_object* with *schema* when using the ON DIRECTORY clause. The object must be a directory. See "CREATE DIRECTORY" on page 7-264.

ON *object*	identifies the object on which the object privileges are revoked. This object can be ■ A table, view, sequence, procedure, stored function, or package, materialized view / snapshot, ■ A synonym for a table, view, sequence, procedure, stored function, package, or materialized view / snapshot ■ A library, indextype, or user-defined operator. If you do not qualify *object* with *schema*, Oracle assumes the object is in your own schema. ■ If you revoke the SELECT object privilege (with or without the GRANT OPTION) on the containing table or snapshot of a materialized view, the materialized view will be invalidated. ■ If you revoke the SELECT object privilege (with or without the GRANT OPTION) on any of the master tables of a materialized view, both the view and its containing table or materialized view will be invalidated.
FROM	identifies users and roles from which the object privileges are revoked. PUBLIC revokes object privileges from all users.
CASCADE CONSTRAINTS	This clause is relevant only if you revoke the REFERENCES privilege or ALL [PRIVILEGES]. It drops any referential integrity constraints that the revokee has defined using the REFERENCES privilege (which might have been granted either explicitly or implicitly through a grant of ALL [PRIVILEGES]).
FORCE	revokes EXECUTE object privilege on user-defined type objects with table or type dependencies. You must use the FORCE clause to revoke the EXECUTE object privilege on user-defined type objects with table dependencies. If you specify FORCE, all privileges will be revoked, but all dependent objects are marked INVALID, data in dependent tables becomes inaccessible, and all dependent function-based indexes are marked UNUSABLE. (Regranting the necessary type privilege will revalidate the table.) For detailed information about type dependencies and user-defined object privileges, see *Oracle8i Concepts*.

Examples

Basic Example You can grant DELETE, INSERT, SELECT, and UPDATE privileges on the table BONUS to the user PEDRO with the following statement:

```
GRANT ALL
    ON bonus TO pedro;
```

To revoke the DELETE privilege on BONUS from PEDRO, issue the following statement:

```
REVOKE DELETE
```

```
    ON bonus FROM pedro;
```

To revoke the remaining privileges on BONUS that you granted to PEDRO, issue the following statement:

```
REVOKE ALL
    ON bonus FROM pedro;
```

PUBLIC Example You can grant SELECT and UPDATE privileges on the view REPORTS to all users by granting the privileges to the role PUBLIC:

```
GRANT SELECT, UPDATE
    ON reports TO public;
```

The following statement revokes UPDATE privilege on REPORTS from all users:

```
REVOKE UPDATE
    ON reports FROM public;
```

Users can no longer update the REPORTS view, although users can still query it. However, if you have also granted UPDATE privilege on REPORTS to any users, either directly or through roles, these users retain the privilege.

Schema Example You can grant the user BLAKE the SELECT privilege on the ESEQ sequence in the schema ELLY with the following statement:

```
GRANT SELECT
    ON elly.eseq TO blake;
```

To revoke the SELECT privilege on ESEQ from BLAKE, issue the following statement:

```
REVOKE SELECT
    ON elly.eseq FROM blake;
```

However, if the user ELLY has also granted SELECT privilege on ESEQ to BLAKE, BLAKE can still use ESEQ by virtue of ELLY's grant.

CASCADE CONSTRAINTS Example You can grant BLAKE the privileges REFERENCES and UPDATE on the EMP table in the schema SCOTT with the following statement:

```
GRANT REFERENCES, UPDATE
    ON scott.emp TO blake;
```

BLAKE can exercise the REFERENCES privilege to define a constraint in his own DEPENDENT table that refers to the EMP table in the schema SCOTT:

```
CREATE TABLE dependent
(dependno NUMBER,
 dependname VARCHAR2(10),
 employee NUMBER
    CONSTRAINT in_emp REFERENCES scott.emp(ename) );
```

You can revoke the REFERENCES privilege on SCOTT.EMP from BLAKE, by issuing the following statement that contains the CASCADE CONSTRAINTS clause:

```
REVOKE REFERENCES
    ON scott.emp
    FROM blake
    CASCADE CONSTRAINTS;
```

Revoking BLAKE's REFERENCES privilege on SCOTT.EMP causes Oracle to drop the IN_EMP constraint, because BLAKE required the privilege to define the constraint.

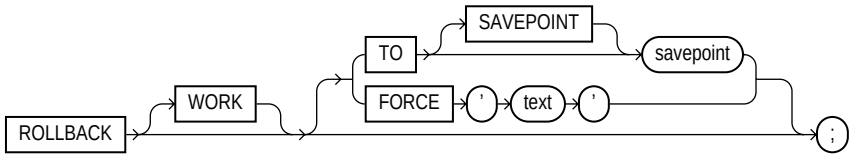
However, if BLAKE has also been granted the REFERENCES privilege on SCOTT.EMP by a user other than you, Oracle does not drop the constraint. BLAKE still has the privilege necessary for the constraint by virtue of the other user's grant.

Directory Example You can revoke READ privilege on directory BFILE_DIR1 from SUE, by issuing the following statement:

```
REVOKE READ ON DIRECTORY bfile_dir1 FROM sue;
```

ROLLBACK

Syntax



Purpose

To undo work done in the current transaction, or to manually undo the work done by an in-doubt distributed transaction. For information on transactions, see *Oracle8i Concepts*. For information on setting characteristics of the current transaction, see "SET TRANSACTION" on page 7-572. See also "SAVEPOINT" on page 7-539.

> **Note:** Oracle recommends that you explicitly end transactions in application programs using either a COMMIT or ROLLBACK statement. If you do not explicitly commit the transaction and the program terminates abnormally, Oracle rolls back the last uncommitted transaction. See also "COMMIT" on page 7-214.

Prerequisites

To roll back your current transaction, no privileges are necessary.

To manually roll back an in-doubt distributed transaction that you originally committed, you must have the FORCE TRANSACTION system privilege. To manually roll back an in-doubt distributed transaction originally committed by another user, you must have the FORCE ANY TRANSACTION system privilege.

Keywords and Parameters

WORK	is optional and is provided for ANSI compatibility.
TO SAVEPOINT *savepoint*	rolls back the current transaction to the specified savepoint. If you omit this clause, the ROLLBACK statement rolls back the entire transaction. See also "SAVEPOINT" on page 7-539.

Using ROLLBACK **without** the TO SAVEPOINT clause performs the following operations:

- Ends the transaction
- Undoes all changes in the current transaction
- Erases all savepoints in the transaction
- Releases the transaction's locks

Using ROLLBACK **with** the TO SAVEPOINT clause performs the following operations:

- Rolls back just the portion of the transaction after the savepoint.
- Erases all savepoints created after that savepoint. The named savepoint is retained, so you can roll back to the same savepoint multiple times. Prior savepoints are also retained.
- Releases all table and row locks acquired since the savepoint. Other transactions that have requested access to rows locked after the savepoint must continue to wait until the transaction is committed or rolled back. Other transactions that have not already requested the rows can request and access the rows immediately.

Restriction: You cannot manually roll back an in-doubt transaction to a savepoint.

FORCE — manually rolls back an in-doubt distributed transaction. The transaction is identified by the '*text*' containing its local or global transaction ID. To find the IDs of such transactions, query the data dictionary view DBA_2PC_PENDING. For more information on distributed transactions and rolling back in-doubt transactions, see *Oracle8i Distributed Database Systems*.

A ROLLBACK statement with a FORCE clause rolls back only the specified transaction. Such a statement does not affect your current transaction.

Restriction: ROLLBACK statements with the FORCE clause are not supported in PL/SQL.

Examples

The following statement rolls back your entire current transaction:

```
ROLLBACK;
```

The following statement rolls back your current transaction to savepoint SP5:

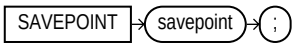
```
ROLLBACK TO SAVEPOINT sp5;
```

The following statement manually rolls back an in-doubt distributed transaction:

```
ROLLBACK WORK
    FORCE '25.32.87';
```

SAVEPOINT

Syntax



Purpose

To identify a point in a transaction to which you can later roll back.

For information on savepoints, see *Oracle8i Concepts*. For information on rolling back transactions, see "ROLLBACK" on page 7-537. For information on setting characteristics of the current transaction, see "SET TRANSACTION" on page 7-572.

Prerequisites

None.

Keywords and Parameters

savepoint	is the name of the savepoint to be created. Savepoint names must be distinct within a given transaction. If you create a second savepoint with the same identifier as an earlier savepoint, the earlier savepoint is erased. After a savepoint has been created, you can either continue processing, commit your work, roll back the entire transaction, or roll back to the savepoint.

Example

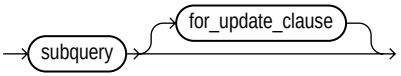
To update BLAKE's and CLARK's salary, check that the total company salary does not exceed 27,000, then reenter CLARK's salary, enter:

```
UPDATE emp
    SET sal = 2000
    WHERE ename = 'BLAKE';
SAVEPOINT blake_sal;

UPDATE emp
    SET sal = 1500
    WHERE ename = 'CLARK';
SAVEPOINT clark_sal;
```

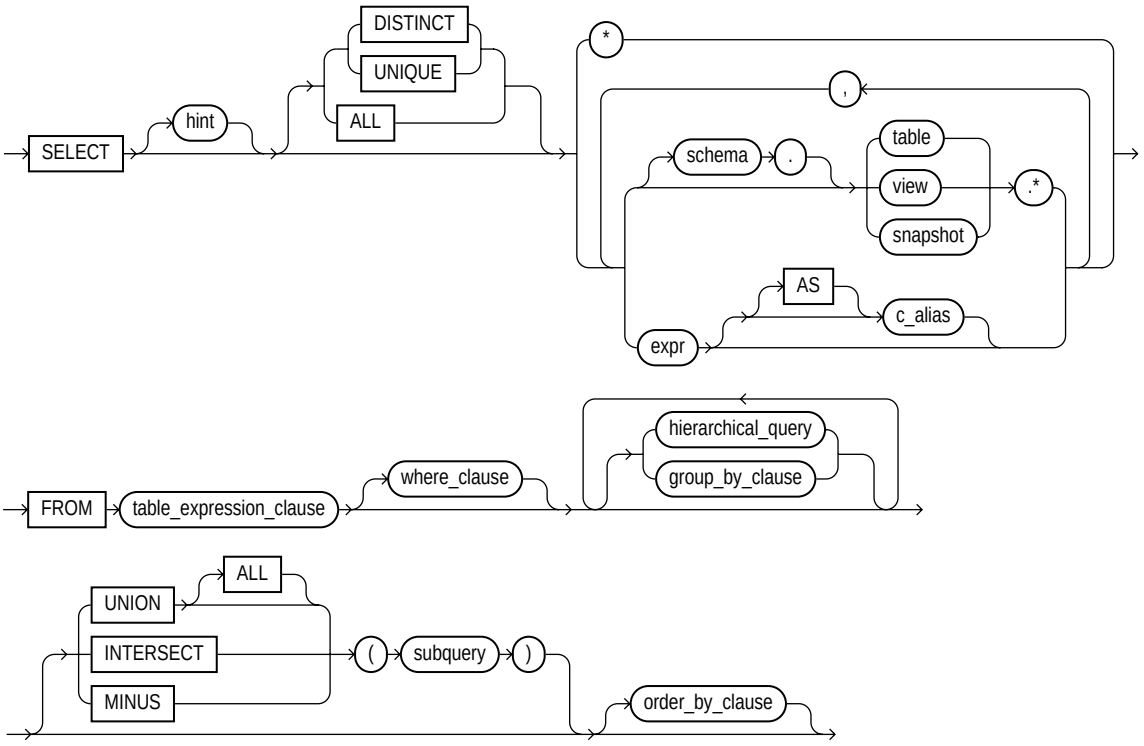
```
SELECT SUM(sal) FROM emp;

ROLLBACK TO SAVEPOINT blake_sal;

UPDATE emp
    SET sal = 1200
    WHERE ename = 'CLARK';

COMMIT;
```

SELECT and Subqueries

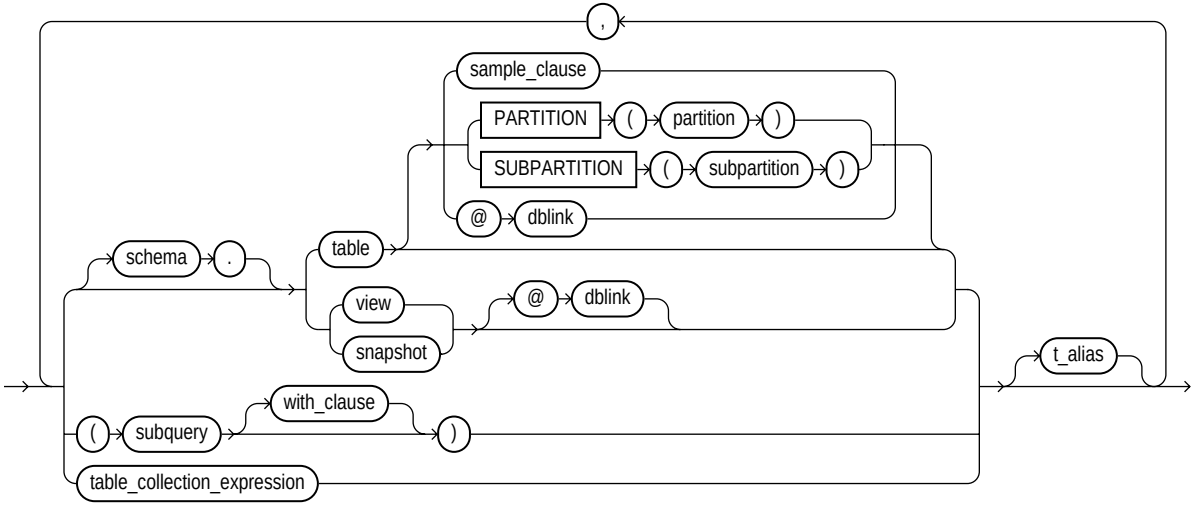
Syntax



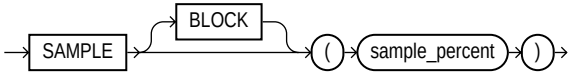
subquery::=



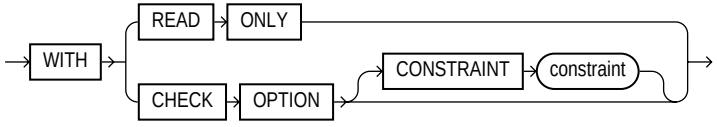
table_expression_clause::=



sample_clause::=

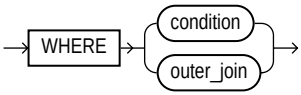


with_clause::=

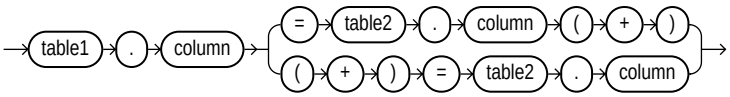


table_collection_expression::=

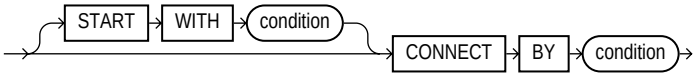
where_clause::=



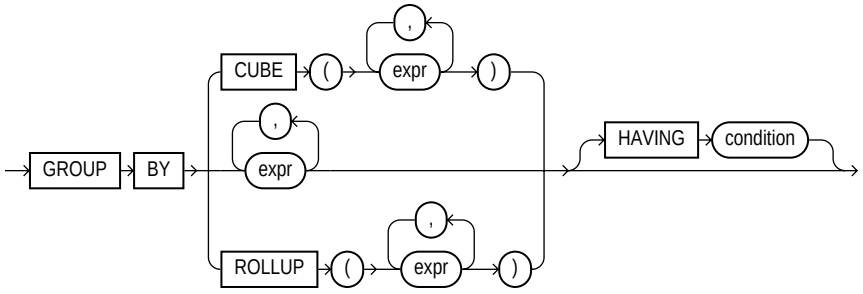
outer_join::=



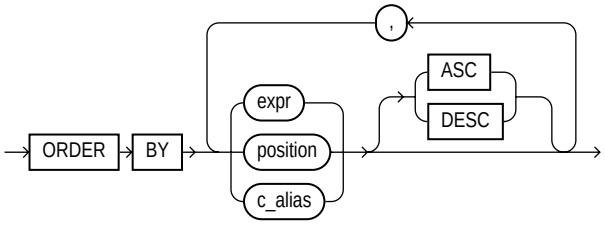
hierarchical_query_clause::=



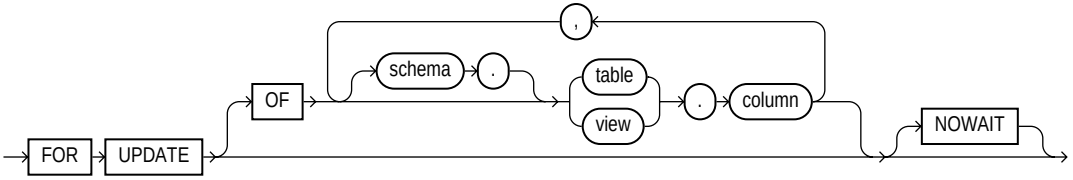
group_by_clause::=



order_by_clause::=



for_update_clause::=



Purpose

To retrieve data from one or more tables, object tables, views, object views, or materialized views. For general information on queries and subqueries, see "Queries and Subqueries" on page 5-18.

> **Note:** If the result (or part of the result) of a SELECT statement is equivalent to an existing materialized view, Oracle may use the materialized view in place of one or more tables specified in the SELECT statement. This substitution is called **query rewrite**, and takes place only if cost optimization is enabled and the QUERY_REWRITE_ENABLED parameter is set to TRUE. To determine whether query write has occurred, use the EXPLAIN PLAN statement (see "EXPLAIN PLAN" on page 7-486). For more information on materialized views and query rewrite, see *Oracle8i Tuning*.

Prerequisites

For you to select data from a table or materialized view, the table or materialized view must be in your own schema or you must have the SELECT privilege on the table or materialized view.

For you to select rows from the base tables of a view,

- You must have the SELECT privilege on the view, and
- Whoever owns the schema containing the view must have the SELECT privilege on the base tables.

The SELECT ANY TABLE system privilege also allows you to select data from any table or any materialized view or any view's base table.

Keywords and Parameters

hint is a comment that passes instructions to the optimizer on choosing an execution plan for the statement. For the syntax and description of hints, see "Hints" on page 2-58 and *Oracle8i Tuning*.

DISTINCT | UNIQUE returns only one copy of each set of duplicate rows selected (these two keywords are synonymous). Duplicate rows are those with matching values for each expression in the select list.

Restrictions:

- When you specify DISTINCT or UNIQUE, the total number of bytes in all select list expressions is limited to the size of a data block minus some overhead. This size is specified by the initialization parameter DB_BLOCK_SIZE.
- You cannot specify DISTINCT if the FROM clause contains LOB columns.

ALL returns all rows selected, including all copies of duplicates. The default is ALL.

* selects all columns from all tables, views, or snapshots listed in the FROM clause.

Note: If you are selecting from a table (that is, you specify *table* in the FROM clause rather than an *view* or a *snapshot*), columns that have been marked as UNUSED by the ALTER TABLE SET UNUSED statement are not selected. See "ALTER TABLE" on page 7-113.

schema is the schema containing the selected table, view, or snapshot. If you omit *schema*, Oracle assumes the table, view, or snapshot is in your own schema.

table. | *view.* | *snapshot.** selects all columns from the specified table, view, or snapshot. You can use the schema qualifier to select from a table, view, or snapshot in a schema other than your own. A query that selects rows from two or more tables, views, or snapshots is a join. For more information, see "Joins" on page 5-21.

expr selects an expression. See the syntax description of *expr* in "Expressions" on page 5-1. A column name in this list can be qualified with *schema* only if the table, view, or snapshot containing the column is qualified with *schema* in the FROM clause.

Restrictions:

- If you also specify a *group_by_clause* in this statement, this select list can contain only the following types of expressions:

  - constants

  - aggregate functions and the functions USER, UID, and SYSDATE

  - expressions identical to those in the *group_by_clause*

  - expressions involving the above expressions that evaluate to the same value for all rows in a group

- You can select a rowid from a join view only if the join has one and only one key-preserved table. The rowid of that table becomes the rowid of the view. For information on key-preserved tables, see *Oracle8i Administrator's Guide*.
- If two or more tables have some column names in common, you must qualify column names with names of tables.

c_alias provides a different name for the column expression and causes the alias to be used in the column heading. The AS keyword is optional. The alias effectively renames the select list item for the duration of the query. The alias can be used in the *order_by_clause*, but not other clauses in the query.

FROM *table_expression_clause* specifies the table, view, snapshot, or partition from which data is selected, or a subquery that specifies the objects from which data is selected.

sample_clause causes Oracle to select from a random sample of rows from the table, rather than from the entire table.

- BLOCK instructs Oracle to perform random block sampling instead of random row sampling. For a discussion of the difference, refer to *Oracle8i Concepts*.
- *sample_percent* is a number specifying the percentage of the total row or block count to be included in the sample. The value must be in the range .000001 to 99.

Restrictions:

- You can specify SAMPLE only in a query that selects from a single table. Joins are not supported.
- When you specify SAMPLE, Oracle automatically uses the cost-based optimizer. The rule-based optimizer is not supported with this clause.

WARNING: The use of statistically incorrect assumptions when using this feature can lead to incorrect or undesirable results. Refer to *Oracle8i Concepts* **for more information on using the** *sample_clause***.**

PARTITION (*partition*) SUBPARTITION (*subpartition*) specifies partition-level data retrieval. The *partition* parameter may be the name of the partition within *table* from which to retrieve data or a more complicated predicate restricting retrieval to just one partition of the table.

dblink is the complete or partial name for a database link to a remote database where the table, view, or snapshot is located. This database need not be an Oracle database.

For more information on referring to database links, see "Referring to Objects in Remote Databases" on page 2-74. For more information about distributed queries, see "Distributed Queries" on page 5-25.

If you omit *dblink*, Oracle assumes that the table, view, or snapshot is on the local database.

table, view, snapshot is the name of a table, view, or snapshot from which data is selected.

with_clause restricts the subquery in one of the following ways:

- WITH READ ONLY specifies that the subquery cannot be updated.
- WITH CHECK OPTION specifies that, if the subquery is used in place of a table in an INSERT, UPDATE, or DELETE statement, Oracle prohibits any changes to that table that would produce rows that are not included in the subquery. See the WITH CHECK OPTION Example on page 7-558.

table_collection_expression informs Oracle that the collection value expression should be treated as a table for purposes of query and DML operations. The *collection_expression* can be a subquery, a column, a CAST or DECODE expression, a function, or a collection constructor. Regardless of its form, it must return a collection value (that is, a value whose type is nested table or varray). This process of extracting the elements of a collection is called **collection unnesting**. See "Collection Unnesting Examples" on page 7-564.

The *collection_expression* can reference columns of tables defined to its left in the FROM clause. This is called **left correlation**. Left correlation can occur only in *table_collection_expression*. Other subqueries cannot contains references to columns defined outside the subquery.

The optional "(+)" lets you specify that *table_collection_expression* should return a row with all fields set to NULL if the collection is null or empty. The "(+)" is valid only if *collection_expression* uses left correlation. The result is similar to that of an outer join. For more information see "Outer Joins" on page 5-22.

Restriction: Queries and subqueries referencing nested tables cannot be parallelized.

Note: In earlier releases of Oracle, when *collection_expression* was a subquery, *table_collection_expr* was expressed as "THE subquery". That usage is now deprecated.

t_alias provides a **correlation name** for the table, view, snapshot, or subquery for evaluating the query and is most often used in a correlated query. Other references to the table, view, or snapshot throughout the query must refer to the alias.

Note: This alias is **required** if the *table_expression_clause* references any object type attributes or object type methods.

where_clause restricts the rows selected to those that satisfy one or more conditions.

- *condition* can be any valid SQL condition. See the syntax description of condition in "Conditions" on page 5-13.
- *outer_join* applies only if the *table_expression_clause* specifies more than one table. This special form of condition requires Oracle to return all the rows that satisfy the condition, as well as all the rows from one of the tables for which no rows of the other table satisfy the condition. For more information, including rules and restrictions that apply to outer joins, see "Outer Joins" on page 5-22.

  If one of the elements in the *table_expression_clause* is actually a nested table or some other form of collection, you specify the outer-join syntax in the *table_collection_expression* rather than in the *where_clause*.

If you omit this clause, Oracle returns all rows from the tables, views, or snapshots in the FROM clause.

Note: If this clause refers to a DATE column of a partitioned table or index, you must specify the year completely using the TO_DATE function with a 4-character format mask. Otherwise Oracle will not perform partition pruning. "PARTITION Example" on page 7-552.

---

hierarchical_query_clause lets you select rows in a hierarchical order. For a discussion of hierarchical queries, see "Hierarchical Queries" on page 5-19.

The preceding *where_clause*, if specified, restricts the rows returned by the query without affecting other rows of the hierarchy.

SELECT statements that contain hierarchical queries can contain the LEVEL pseudocolumn. LEVEL returns the value 1 for a root node, 2 for a child node of a root node, 3 for a grandchild, etc. The number of levels returned by a hierarchical query may be limited by available user memory.

For more information on LEVEL, see the section "Pseudocolumns" on page 2-51.

Restrictions: If you specify a hierarchical query:

- The same statement cannot also perform a join.
- The same statement cannot also select data from a view whose query performs a join.
- If you also specify the *order_by_clause*, it takes precedence over any ordering specified by the hierarchical query.

START WITH identifies the row(s) to be used as the root(s) of a hierarchical query. This clause specifies a condition that the roots must satisfy. If you omit this clause, Oracle uses all rows in the table as root rows. The START WITH *condition* can contain a subquery.

CONNECT BY specifies the relationship between parent rows and child rows of the hierarchy. *condition* can be any condition as described in "Conditions" on page 5-13. However, some part of the condition must use the PRIOR operator to refer to the parent row. The part of the condition containing the PRIOR operator must have one of the following forms:

```
PRIOR expr comparison_operator expr
expr comparison_operator PRIOR expr
```

Restriction: The CONNECT BY condition cannot contain a subquery.

group_by_clause groups the selected rows based on the value of *expr(s)* for each row, and returns a single row of summary information for each group. If this clause contains CUBE or ROLLUP extensions, then superaggregate groupings are produced in addition to the regular groupings.

Expressions in the *group_by_clause* can contain any columns in the tables, views, and snapshots in the FROM clause, regardless of whether the columns appear in the select list.

Restrictions:

- The *group_by_clause* can contain no more than 255 expressions.
- You cannot specify LOB columns, nested tables, or varrays as part of *expr*.
- The total number of bytes in all expressions in the *group_by_clause* is limited to the size of a data block (specified by the initialization parameter DB_BLOCK_SIZE) minus some overhead.
- If the *group_by_clause* references any object columns, the query will not be parallelized.

ROLLUP is an extension to the *group_by_clause* that groups the selected rows based on the values of the first n, n-1, n-2, ... 0 expressions for each row, and returns a single row of summary for each group. You can use the ROLLUP operation to produce **subtotal values**.

For example, given three expressions in the ROLLUP clause of the *group_by_clause*, the operation results in n+1 = 3+1 = 4 groupings.

Rows based on the values of the first 'n' expressions are called **regular rows**, and the others are called **superaggregate rows**.

An example appears with the description of the GROUPING function. See "GROUPING" on page 4-16. See also *Oracle8i Application Developer's Guide - Fundamentals.*

CUBE is an extension to the *group_by_clause* that groups the selected rows based on the values of all possible combinations of expressions for each row, and returns a single row of summary information for each group. You can use the CUBE operation to produce **cross-tabulation values**.

For example, given three expressions in the CUBE clause of the *group_by_clause*, the operation results in $2^n = 2^3 = 8$ groupings. Rows based on the values of 'n' expressions are called regular rows, and the rest are called **superaggregate rows**.

See the "CUBE Example" on page 7-553 and "GROUPING" on page 4-16. See also *Oracle8i Application Developer's Guide - Fundamentals.*

HAVING — restricts the groups of rows returned to those groups for which the specified *condition* is TRUE. If you omit this clause, Oracle returns summary rows for all groups.

Specify GROUP BY and HAVING after the *where_clause* and CONNECT BY clause. If you specify both GROUP BY and HAVING, they can appear in either order.

See also the syntax description of *expr* in "Expressions" on page 5-1 and the syntax description of *condition* in "Conditions" on page 5-13.

UNION | UNION ALL | INTERSECT | MINUS — are set operators that combine the rows returned by two SELECT statements into a single result. The number and datatypes of the columns selected by each component query must be the same, but the column lengths can be different.

If you combine more than two queries with set operators, Oracle evaluates adjacent queries from left to right. You can use parentheses to specify a different order of evaluation.

For information on these operators, see "Set Operators" on page 3-12.

Restrictions:

- These set operators are not valid on columns of type BLOB, CLOB, BFILE, varray, or nested table.
- To reference a column, you must use an alias to name the column.
- You cannot also specify the *for_update_clause* with these set operators.
- You cannot specify the *order_by_clause* in the *subquery* of these operators.
- You cannot use these operators in SELECT statements containing TABLE collection expressions.
- The total number of bytes in all select list expressions of a component query is limited to the size of a data block (specified by the initialization parameter DB_BLOCK_SIZE) minus some overhead.

Note: To comply with emerging SQL standards, a future release of Oracle will give the INTERSECT operator greater precedence than the other set operators. Therefore, you should use parentheses to specify order of evaluation in queries that use the INTERSECT operator with other set operators.

order_by_clause orders rows returned by the statement. Without an *order_by_clause*, no guarantee exists that the same query executed more than once will retrieve rows in the same order. For a discussion of ordering query results, see "Sorting Query Results" on page 5-20.

- *expr* orders rows based on their value for *expr*. The expression is based on columns in the select list or columns in the tables, views, or snapshots in the FROM clause.
- *position* orders rows based on their value for the expression in this position of the select list; *position* must be an integer.
- ASC and DESC specify either ascending or descending order. ASC is the default.

You can specify multiple expressions in the *order_by_clause*. Oracle first sorts rows based on their values for the first expression. Rows with the same value for the first expression are then sorted based on their values for the second expression, and so on. Oracle sorts nulls following all others in ascending order and preceding all others in descending order.

Restrictions:

- If you have specified the DISTINCT operator in this statement, this clause cannot refer to columns unless they appear in the select list.
- An *order_by_clause* can contain no more than 255 expressions.
- You cannot order by a LOB column, nested table, or varray.

If you specify a *group_by_clause* in the same statement, this *order_by_clause* is restricted to the following expressions:

- Constants
- Aggregate functions
- The functions USER, UID, and SYSDATE
- Expressions identical to those in the *group_by_clause*
- Expressions involving the above expressions that evaluate to the same value for all rows in a group.

for_update_clause locks the selected rows so that other users cannot lock or update the rows until you end your transaction. You can specify this clause only in a top-level SELECT statement (not in subqueries).

- Prior to updating a LOB value, you must lock the row containing the LOB. One way to lock the row is with a SELECT... FOR UPDATE statement. See "LOB Locking Example" on page 7-556.
- Nested table rows are not locked as a result of locking the parent table rows. If you want the nested table rows to be locked, you must lock them explicitly.

OF — Locks the select rows only for a particular table in a join. The columns in the OF clause only specify which tables' rows are locked. The specific columns of the table that you specify are not significant. If you omit this clause, Oracle locks the selected rows from all the tables in the query.

NOWAIT	returns control to you if the SELECT statement attempts to lock a row that is locked by another user. If you omit this clause, Oracle waits until the row is available and then returns the results of the SELECT statement.

Restrictions:

- You cannot specify this clause with the following other constructs: DISTINCT operator, *group_by_clause*, set operators, aggregate functions, or the CURSOR operator.
- The tables locked by this clause must all be located on the same database, and on the same database as any LONG columns and sequences referenced in the same statement.

Examples

Simple Query Examples The following statement selects rows from the EMP table with the department number of 30:

```
SELECT *
    FROM emp
    WHERE deptno = 30;
```

The following statement selects the name, job, salary and department number of all employees except sales people from department number 30:

```
SELECT ename, job, sal, deptno
    FROM emp
    WHERE NOT (job = 'SALESMAN' AND deptno = 30);
```

The following statement selects from subqueries in the FROM clause and gives departments' total employees and salaries as a decimal value of all the departments:

```
SELECT a.deptno "Department",
       a.num_emp/b.total_count "%Employees",
       a.sal_sum/b.total_sal "%Salary"
  FROM
 (SELECT deptno, COUNT(*) num_emp, SUM(SAL) sal_sum
    FROM scott.emp
    GROUP BY deptno) a,
 (SELECT COUNT(*) total_count, SUM(sal) total_sal
    FROM scott.emp) b ;
```

PARTITION Example You can select rows from a single partition of a partitioned table by specifying the keyword PARTITION in the FROM clause. This SQL statement assigns an alias for and retrieves rows from the NOV98 partition of the SALES table:

```
SELECT * FROM sales PARTITION (nov98) s
   WHERE s.amount_of_sale > 1000;
```

The following example selects rows from the SALES table for sales earlier than a specified date:

```
SELECT * FROM sales
   WHERE sale_date < TO_DATE('1998-06-15', 'YYYY-MM-DD');
```

SAMPLE example: The following query estimates the number of employees in the EMP table:

```
SELECT COUNT(*) * 100 FROM emp SAMPLE BLOCK (1);
```

GROUP BY Examples To return the minimum and maximum salaries for each department in the employee table, issue the following statement:

```
SELECT deptno, MIN(sal), MAX (sal)
     FROM emp
     GROUP BY deptno;

DEPTNO     MIN(SAL)   MAX(SAL)
---------- ---------- ----------
        10       1300       5000
        20        800       3000
        30        950       2850
```

To return the minimum and maximum salaries for the clerks in each department, issue the following statement:

```
SELECT deptno, MIN(sal), MAX (sal)
     FROM emp
     WHERE job = 'CLERK'
     GROUP BY deptno;

DEPTNO     MIN(SAL)   MAX(SAL)
---------- ---------- ----------
        10       1300      1300
        20        800      1100
        30        950       950
```

CUBE Example To return the number of employees and their average yearly salary across all possible combinations of department and job category, issue the following query:

```
SELECT DECODE(GROUPING(dname), 1, 'All Departments',
      dname) AS dname,
   DECODE(GROUPING(job), 1, 'All Jobs', job) AS job,
   COUNT(*) "Total Empl", AVG(sal) * 12 "Average Sal"
   FROM emp, dept
   WHERE dept.deptno = emp.deptno
   GROUP BY CUBE (dname, job);

DNAME           JOB       Total Empl Average Sa
--------------- --------- ---------- ----------
ACCOUNTING      CLERK              1      15600
ACCOUNTING      MANAGER            1      29400
ACCOUNTING      PRESIDENT          1      60000
ACCOUNTING      All Jobs           3      35000
RESEARCH        ANALYST            2      36000
RESEARCH        CLERK              2      11400
RESEARCH        MANAGER            1      35700
RESEARCH        All Jobs           5      26100
SALES           CLERK              1      11400
SALES           MANAGER            1      34200
SALES           SALESMAN           4      16800
SALES           All Jobs           6      18800
All Departments ANALYST            2      36000
All Departments CLERK              4      12450
All Departments MANAGER            3      33100
All Departments PRESIDENT          1      60000
All Departments SALESMAN           4      16800
All Departments All Jobs          14 24878.5714
```

Hierarchical Query Examples The following CONNECT BY clause defines a hierarchical relationship in which the EMPNO value of the parent row is equal to the MGR value of the child row:

```
CONNECT BY PRIOR empno = mgr;
```

In the following CONNECT BY clause, the PRIOR operator applies only to the EMPNO value. To evaluate this condition, Oracle evaluates EMPNO values for the parent row and MGR, SAL, and COMM values for the child row:

```
CONNECT BY PRIOR empno = mgr AND sal > comm;
```

To qualify as a child row, a row must have a MGR value equal to the EMPNO value of the parent row and it must have a SAL value greater than its COMM value.

HAVING Example To return the minimum and maximum salaries for the clerks in each department whose lowest salary is below $1,000, issue the next statement:

```
SELECT deptno, MIN(sal), MAX (sal)
     FROM emp
     WHERE job = 'CLERK'
     GROUP BY deptno
     HAVING MIN(sal) < 1000;

DEPTNO     MIN(SAL)   MAX(SAL)
---------- ---------- ----------
        20        800       1100
        30        950        950
```

ORDER BY Examples To select all salesmen's records from EMP, and order the results by commission in descending order, issue the following statement:

```
SELECT *
    FROM emp
    WHERE job = 'SALESMAN'
    ORDER BY comm DESC;
```

To select the employees from EMP ordered first by ascending department number and then by descending salary, issue the following statement:

```
SELECT ename, deptno, sal
    FROM emp
    ORDER BY deptno ASC, sal DESC;
```

To select the same information as the previous SELECT and use the positional ORDER BY notation, issue the following statement:

```
SELECT ename, deptno, sal
    FROM emp
    ORDER BY 2 ASC, 3 DESC;
```

FOR UPDATE Examples The following statement locks rows in the EMP table with clerks located in New York and locks rows in the DEPT table with departments in New York that have clerks:

```
SELECT empno, sal, comm
    FROM emp, dept
    WHERE job = 'CLERK'
        AND emp.deptno = dept.deptno
        AND loc = 'NEW YORK'
    FOR UPDATE;
```

The following statement locks only those rows in the EMP table with clerks located in New York. No rows are locked in the DEPT table:

```
SELECT empno, sal, comm
    FROM emp, dept
    WHERE job = 'CLERK'
        AND emp.deptno = dept.deptno
        AND loc = 'NEW YORK'
    FOR UPDATE OF emp.sal;
```

LOB Locking Example The following example uses a SELECT ... FOR UPDATE statement to lock a row containing a LOB prior to updating the LOB value.

```
INSERT INTO t_table VALUES (1, 'abcd');

COMMIT;
   DECLARE
     num_var      NUMBER;
     clob_var     CLOB;
     clob_locked  CLOB;
     write_amount NUMBER;
     write_offset NUMBER;
     buffer       VARCHAR2(20) := 'efg';

   BEGIN
     SELECT clob_col INTO clob_locked FROM t_table
     WHERE num_col = 1 FOR UPDATE;

  write_amount := 3;
  dbms_lob.write(clob_locked, write_amount, write_offset, buffer);
END;
```

Table Collection Examples You can perform DML operations on nested tables only if they are defined as columns of a table. Therefore, when the *table_expression_clause* of an INSERT, DELETE, or UPDATE statement is a *table_collection_expression*, the collection expression must be a subquery that selects the table's nested table column. The examples that follow are based on this scenario:

```
   CREATE TYPE ProjectType AS OBJECT(
       pno   NUMBER,
       pname CHAR(31),
       budget NUMBER);
   CREATE TYPE ProjectSet AS TABLE OF ProjectType;
```

```
CREATE TABLE Dept (dno NUMBER, dname CHAR(31), projs ProjectSet)
    NESTED TABLE projs STORE AS
        ProjectSetTable ((Primary Key(Nested_Table_Id, pno))
ORGANIZATION
INDEX COMPRESS 1);

INSERT INTO Dept VALUES (1, 'Engineering', ProjectSet());
```

This example inserts into the 'Engineering' department's 'projs' nested table:

```
INSERT INTO TABLE(SELECT d.projs
                  FROM   Dept d
                  WHERE  d.dno = 1)
  VALUES (1, 'Collection Enhancements', 10000);
```

This example updates the 'Engineering' department's 'projs' nested table:

```
UPDATE TABLE(SELECT d.projs
             FROM   Dept d
             WHERE  d.dno = 1) p
  SET  p.budget = p.budget + 1000;
```

This example deletes from the 'Engineering' department's 'projs' nested table

```
DELETE TABLE(SELECT d.projs
             FROM   Dept d
             WHERE  d.dno = 1) p
 WHERE p.budget > 100000;
```

Subquery Examples To determine who works in Taylor's department, issue the following statement:

```
SELECT ename, deptno
    FROM emp
    WHERE deptno =
        (SELECT deptno
            FROM emp
            WHERE ename = 'TAYLOR');
```

To give all employees in the EMP table a 10% raise if they have not already been issued a bonus (if they do not appear in the BONUS table), issue the following statement:

```
UPDATE emp
    SET sal = sal * 1.1
    WHERE empno NOT IN (SELECT empno FROM bonus);
```

To create a duplicate of the DEPT table named NEWDEPT, issue the following statement:

```
CREATE TABLE newdept (deptno, dname, loc)
    AS SELECT deptno, dname, loc FROM dept;
```

WITH CHECK OPTION Example The following statement is legal even though the second value violates the condition of the subquery *where_clause*:

```
INSERT INTO
   (SELECT ename, deptno FROM emp WHERE deptno < 10)
    VALUES ('Taylor', 20);
```

However, the following statement is illegal because of the WITH CHECK OPTION clause:

```
INSERT INTO
   (SELECT ename, deptno FROM emp
       WHERE deptno < 10
       WITH CHECK OPTION)
    VALUES ('Taylor', 20);
```

Equijoin Examples This equijoin returns the name and job of each employee and the number and name of the department in which the employee works:

```
SELECT ename, job, dept.deptno, dname
     FROM emp, dept
     WHERE emp.deptno = dept.deptno;

ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
CLARK      MANAGER           10 ACCOUNTING
KING       PRESIDENT         10 ACCOUNTING
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
FORD       ANALYST           20 RESEARCH
SCOTT      ANALYST           20 RESEARCH
JONES      MANAGER           20 RESEARCH
ALLEN      SALESMAN          30 SALES
BLAKE      MANAGER           30 SALES
MARTIN     SALESMAN          30 SALES
JAMES      CLERK             30 SALES
TURNER     SALESMAN          30 SALES
WARD       SALESMAN          30 SALES
```

You must use a join to return this data because employee names and jobs are stored in a different table than department names. Oracle combines rows of the two tables according to this join condition:

```
emp.deptno = dept.deptno
```

The following equijoin returns the name, job, department number, and department name of all clerks:

```
SELECT ename, job, dept.deptno, dname
     FROM emp, dept
     WHERE emp.deptno = dept.deptno
     AND job = 'CLERK';

ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
JAMES      CLERK             30 SALES
```

This query is identical to the preceding example, except that it uses an additional *where_clause* condition to return only rows with a JOB value of 'CLERK'.

Self Join Example The following query uses a self join to return the name of each employee along with the name of the employee's manager:

```
SELECT e1.ename||' works for '||e2.ename
"Employees and their Managers"
    FROM emp e1, emp e2   WHERE e1.mgr = e2.empno;

Employees and their Managers
------------------------------
BLAKE works for KING
CLARK works for KING
JONES works for KING
FORD works for JONES
SMITH works for FORD
ALLEN works for BLAKE
WARD works for BLAKE
MARTIN works for BLAKE
SCOTT works for JONES
TURNER works for BLAKE
ADAMS works for SCOTT
```

```
JAMES works for BLAKE
MILLER works for CLARK
```

The join condition for this query uses the aliases E1 and E2 for the EMP table:

```
e1.mgr = e2.empno
```

Outer Join Examples This query uses an outer join to extend the results of the Equijoin example above:

```
SELECT ename, job, dept.deptno, dname
    FROM emp, dept
    WHERE emp.deptno (+) = dept.deptno;

ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
CLARK      MANAGER           10 ACCOUNTING
KING       PRESIDENT         10 ACCOUNTING
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
FORD       ANALYST           20 RESEARCH
SCOTT      ANALYST           20 RESEARCH
JONES      MANAGER           20 RESEARCH
ALLEN      SALESMAN          30 SALES
BLAKE      MANAGER           30 SALES
MARTIN     SALESMAN          30 SALES
JAMES      CLERK             30 SALES
TURNER     SALESMAN          30 SALES
WARD       SALESMAN          30 SALES
                             40 OPERATIONS
```

In this outer join, Oracle returns a row containing the OPERATIONS department even though no employees work in this department. Oracle returns NULL in the ENAME and JOB columns for this row. The join query in this example selects only departments that have employees.

The following query uses an outer join to extend the results of the preceding example:

```
SELECT ename, job, dept.deptno, dname
    FROM emp, dept
    WHERE emp.deptno (+) = dept.deptno
        AND job (+) = 'CLERK';
```

```
ENAME      JOB       DEPTNO     DNAME
---------- --------- ---------- --------------
MILLER     CLERK             10 ACCOUNTING
SMITH      CLERK             20 RESEARCH
ADAMS      CLERK             20 RESEARCH
JAMES      CLERK             30 SALES
                             40 OPERATIONS
```

In this outer join, Oracle returns a row containing the OPERATIONS department even though no clerks work in this department. The (+) operator on the JOB column ensures that rows for which the JOB column is NULL are also returned. If this (+) were omitted, the row containing the OPERATIONS department would not be returned because its JOB value is not 'CLERK'.

This example shows four outer join queries on the CUSTOMERS, ORDERS, LINEITEMS, and PARTS tables. These tables are shown here:

```
SELECT custno, custname
    FROM customers;

CUSTNO     CUSTNAME
---------- --------------------
1          Angelic Co.
2          Believable Co.
3          Cabels R Us

SELECT orderno, custno,
    TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE"
    FROM orders;

ORDERNO    CUSTNO     ORDERDATE
---------- ---------- -----------
      9001          1 OCT-13-1998
      9002          2 OCT-13-1998
      9003          1 OCT-20-1998
      9004          1 OCT-27-1998
      9005          2 OCT-31-1998

SELECT orderno, lineno, partno, quantity
    FROM lineitems;

ORDERNO    LINENO     PARTNO     QUANTITY
---------- ---------- ---------- ----------
      9001          1        101         15
      9001          2        102         10
```

```
      9002          1        101         25
      9002          2        103         50
      9003          1        101         15
      9004          1        102         10
      9004          2        103         20

SELECT partno, partname
    FROM parts;

PARTNO PARTNAME
------ --------
   101 X-Ray Screen
   102 Yellow Bag
   103 Zoot Suit
```

The customer Cables R Us has placed no orders, and order number 9005 has no line items.

The following outer join returns all customers and the dates they placed orders. The (+) operator ensures that customers who placed no orders are also returned:

```
SELECT custname, TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE"
    FROM customers, orders
    WHERE customers.custno = orders.custno (+);

CUSTNAME             ORDERDATE
-------------------- --------------
Angelic Co.          OCT-13-1993
Angelic Co.          OCT-20-1993
Angelic Co.          OCT-27-1993
Believable Co.       OCT-13-1993
Believable Co.       OCT-31-1993
Cables R Us
```

The following outer join builds on the result of the previous one by adding the LINEITEMS table to the FROM clause, columns from this table to the select list, and a join condition joining this table to the ORDERS table to the *where_clause*. This query joins the results of the previous query to the LINEITEMS table and returns all customers, the dates they placed orders, and the part number and quantity of each part they ordered. The first (+) operator serves the same purpose as in the previous query. The second (+) operator ensures that orders with no line items are also returned:

```
SELECT custname,
   TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE",
```

```
   partno,
   quantity
      FROM customers, orders, lineitems
      WHERE customers.custno = orders.custno (+)
      AND orders.orderno = lineitems.orderno (+);

CUSTNAME             ORDERDATE          PARTNO   QUANTITY
-------------------- -------------- ---------- ----------
Angelic Co.          OCT-13-1993           101         15
Angelic Co.          OCT-13-1993           102         10
Angelic Co.          OCT-20-1993           101         15
Angelic Co.          OCT-27-1993           102         10
Angelic Co.          OCT-27-1993           103         20
Believable Co.       OCT-13-1993           101         25
Believable Co.       OCT-13-1993           103         50
Believable Co.       OCT-31-1993
Cables R Us
```

The following outer join builds on the result of the previous one by adding the PARTS table to the FROM clause, the PARTNAME column from this table to the select list, and a join condition joining this table to the LINEITEMS table to the *where_clause*. This query joins the results of the previous query to the PARTS table to return all customers, the dates they placed orders, and the quantity and name of each part they ordered. The first two (+) operators serve the same purposes as in the previous query. The third (+) operator ensures that rows with NULL part numbers are also returned:

```
SELECT custname, TO_CHAR(orderdate, 'MON-DD-YYYY') "ORDERDATE",
    quantity, partname
    FROM customers, orders, lineitems, parts
    WHERE customers.custno = orders.custno (+)
    AND orders.orderno = lineitems.orderno (+)
    AND lineitems.partno = parts.partno (+);

CUSTNAME             ORDERDATE        QUANTITY PARTNAME
-------------------- -------------- ---------- ------------
Angelic Co.          OCT-13-1993            15 X-Ray Screen
Angelic Co.          OCT-13-1993            10 Yellow Bag
Angelic Co.          OCT-20-1993            15 X-Ray Screen
Angelic Co.          OCT-27-1993            10 Yellow Bag
Angelic Co.          OCT-27-1993            20 Zoot Suit
Believable Co.       OCT-13-1993            25 X-Ray Screen
Believable Co.       OCT-13-1993            50 Zoot Suit
Believable Co.       OCT-31-1993
Cables R Us
```

Collection Unnesting Examples Suppose the database contains a table HR_INFO with columns DEPT, LOCATION, and MGR, and a column of nested table type PEOPLE which has NAME, DEPT, and SAL columns. You could get all the rows from HR_INFO and all the rows from PEOPLE using the following statement:

```
SELECT t1.dept, t2.* FROM hr_info t1, TABLE(t1.people) t2
   WHERE t2.dept = t1.dept;
```

Now suppose that PEOPLE is not a nested table column of HR_INFO, but is instead a separate table with columns NAME, DEPT, ADDRESS, HIREDATE, and SAL. You can extract the same rows as in the preceding example with this statement:

```
SELECT t1.department, t2.*
   FROM hr_info t1, TABLE(CAST(MULTISET(
      SELECT t3.name, t3.dept, t3.sal FROM people t3
      WHERE t3.dept = t1.dept)
      AS NESTED_PEOPLE)) t2;
```

Finally suppose that PEOPLE is neither a nested table column of table HR_INFO nor a table itself. Instead, you have created a function PEOPLE_FUNC that extracts from various sources the name, department, and salary of all employees. You can get the same information as in the preceding examples with the following query:

```
SELECT t1.dept, t2.* FROM HY_INFO t1, TABLE(CAST
   (people_func( ... ) AS NESTED_PEOPLE)) t2;
```

For more examples of collection unnesting, see *Oracle8i Application Developer's Guide - Fundamentals.*

LEVEL Examples The following statement returns all employees in hierarchical order. The root row is defined to be the employee whose job is 'PRESIDENT'. The child rows of a parent row are defined to be those who have the employee number of the parent row as their manager number.

```
SELECT LPAD(' ',2*(LEVEL-1)) || ename org_chart,
        empno, mgr, job
    FROM emp
    START WITH job = 'PRESIDENT'
    CONNECT BY PRIOR empno = mgr;

ORG_CHART    EMPNO      MGR        JOB
------------ ---------- ---------- ---------
KING               7839            PRESIDENT
  JONES            7566       7839 MANAGER
```

```
    SCOTT          7788       7566 ANALYST
      ADAMS        7876       7788 CLERK
    FORD           7902       7566 ANALYST
      SMITH        7369       7902 CLERK
  BLAKE            7698       7839 MANAGER
    ALLEN          7499       7698 SALESMAN
    WARD           7521       7698 SALESMAN
    MARTIN         7654       7698 SALESMAN
    TURNER         7844       7698 SALESMAN
      JAMES        7900       7698 CLERK
 CLARK             7782       7839 MANAGER
   MILLER          7934       7782 CLERK
```

The following statement is similar to the previous one, except that it does not select employees with the job 'ANALYST'.

```
SELECT LPAD(' ', 2*(LEVEL-1)) || ename org_chart,
     empno, mgr, job
     FROM emp
     WHERE job != 'ANALYST'
     START WITH job = 'PRESIDENT'
     CONNECT BY PRIOR empno = mgr;

ORG_CHART            EMPNO      MGR         JOB
-------------------- ---------- ---------- ---------
KING                       7839            PRESIDENT
  JONES                    7566       7839 MANAGER
      ADAMS                7876       7788 CLERK
      SMITH                7369       7902 CLERK
  BLAKE                    7698       7839 MANAGER
    ALLEN                  7499       7698 SALESMAN
    WARD                   7521       7698 SALESMAN
    MARTIN                 7654       7698 SALESMAN
    TURNER                 7844       7698 SALESMAN
    JAMES                  7900       7698 CLERK
  CLARK                    7782       7839 MANAGER
    MILLER                 7934       7782 CLERK
```

Oracle does not return the analysts SCOTT and FORD, although it does return employees who are managed by SCOTT and FORD.

The following statement is similar to the first one, except that it uses the LEVEL pseudocolumn to select only the first two levels of the management hierarchy:

```
SELECT LPAD(' ',2*(LEVEL-1)) || ename org_chart,
empno, mgr, job
```

```
    FROM emp
    START WITH job = 'PRESIDENT'
    CONNECT BY PRIOR empno = mgr AND LEVEL <= 2;

ORG_CHART    EMPNO      MGR        JOB
------------ ---------- ---------- ---------
KING               7839            PRESIDENT
  JONES            7566       7839 MANAGER
  BLAKE            7698       7839 MANAGER
  CLARK            7782       7839 MANAGER
```

Distributed Query Example This example shows a query that joins the DEPT table on the local database with the EMP table on the HOUSTON database:

```
SELECT ename, dname
    FROM emp@houston, dept
    WHERE emp.deptno = dept.deptno;
```

Correlated Subquery Examples The following examples show the general syntax of a correlated subquery:

```
SELECT select_list
    FROM table1 t_alias1
    WHERE expr operator
        (SELECT column_list
            FROM table2 t_alias2
            WHERE t_alias1.column
               operator t_alias2.column);
UPDATE table1 t_alias1
    SET column =
        (SELECT expr
            FROM table2 t_alias2
            WHERE t_alias1.column = t_alias2.column);
DELETE FROM table1 t_alias1
    WHERE column operator
        (SELECT expr
            FROM table2 t_alias2
            WHERE t_alias1.column = t_alias2.column);
```

The following statement returns data about employees whose salaries exceed their department average. The following statement assigns an alias to EMP, the table containing the salary information, and then uses the alias in a correlated subquery:

```
SELECT deptno, ename, sal
    FROM emp x
    WHERE sal > (SELECT AVG(sal)
```

```
        FROM emp
        WHERE x.deptno = deptno)
ORDER BY deptno;
```

For each row of the EMP table, the parent query uses the correlated subquery to compute the average salary for members of the same department. The correlated subquery performs the following steps for each row of the EMP table:

1. The DEPTNO of the row is determined.
2. The DEPTNO is then used to evaluate the parent query.
3. If that row's salary is greater than the average salary for that row's department, then the row is returned.

The subquery is evaluated once for each row of the EMP table.

DUAL Table Example The following statement returns the current date:

```
SELECT SYSDATE FROM DUAL;
```

You could select SYSDATE from the EMP table, but Oracle would return 14 rows of the same SYSDATE, one for every row of the EMP table. Selecting from DUAL is more convenient.

Sequence Examples The following statement increments the ZSEQ sequence and returns the new value:

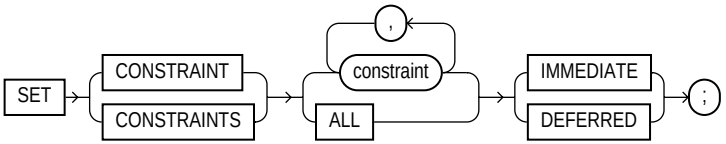
```
SELECT zseq.nextval
    FROM dual;
```

The following statement selects the current value of ZSEQ:

```
SELECT zseq.currval
    FROM dual;
```

SET CONSTRAINT(S)

Syntax



Purpose

To specify, for a particular transaction, whether a deferrable constraint is checked following each DML statement or when the transaction is committed.

Prerequisites

To specify when a deferrable constraint is checked, you must have SELECT privilege on the table to which the constraint is applied unless the table is in your schema.

Keywords and Parameters

constraint	is the name of one or more integrity constraints.
ALL	sets all deferrable constraints for this transaction.
IMMEDIATE	indicates that the conditions specified by the deferrable constraint are checked immediately after each DML statement.
DEFERRED	indicates that the conditions specified by the deferrable constraint are checked when the transaction is committed.

You can verify the success of deferrable constraints prior to committing them by issuing a SET CONSTRAINTS ALL IMMEDIATE statement.

Examples

The following statement sets all deferrable constraints in this transaction to be checked immediately following each DML statement:

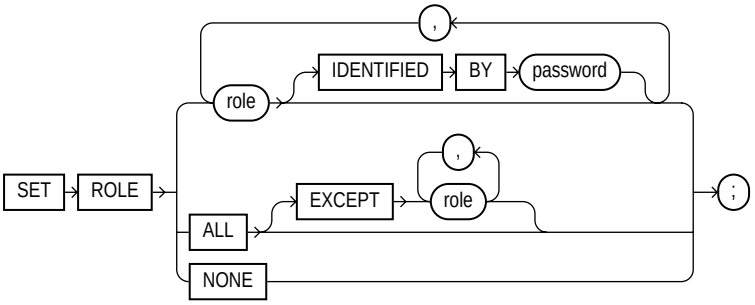
```
SET CONSTRAINTS ALL IMMEDIATE;
```

The following statement checks three deferred constraints when the transaction is committed:

```
SET CONSTRAINTS unq_name, scott.nn_sal,
  adams.pk_dept@dblink DEFERRED;
```

SET ROLE

Syntax



Purpose

To enable and disable roles for your current session. For information on creating roles, see "CREATE ROLE" on page 7-344.

When a user logs on, Oracle enables all privileges granted explicitly to the user and all privileges in the user's default roles. During the session, the user or an application can use the SET ROLE statement any number of times to change the roles currently enabled for the session. The number of roles that can be concurrently enabled is limited by the initialization parameter MAX_ENABLED_ROLES. For information on changing a user's default roles, see "ALTER USER" on page 7-179

You can see which roles are currently enabled by examining the SESSION_ROLES data dictionary view.

Prerequisites

You must already have been granted the roles that you name in the SET ROLE statement.

Keywords and Parameters

role	is a role to be enabled for the current session. Any roles not listed are disabled for the current session. **Restriction:** You cannot specify a role unless it was granted to you either directly or through other roles. IDENTIFIED BY *password* — is the password for a role. If the role has a password, you must specify the password to enable the role.
ALL	enables all roles granted to you for the current session except those optionally listed in the EXCEPT clause. **Restriction:** You cannot use this clause to enable roles with passwords that have been granted directly to you. EXCEPT — Roles listed in the EXCEPT clause must be roles granted directly to you. They cannot be roles granted to you through other roles. If you list a role in the EXCEPT clause that has been granted to you both directly and through another role, the role remains enabled by virtue of the role to which it has been granted.
NONE	disables all roles for the current session, including the DEFAULT role.

Examples

To enable the role GARDENER identified by the password MARIGOLDS for your current session, issue the following statement:

```
SET ROLE gardener IDENTIFIED BY marigolds;
```

To enable all roles granted to you for the current session, issue the following statement:

```
SET ROLE ALL;
```

To enable all roles granted to you except BANKER, issue the following statement:

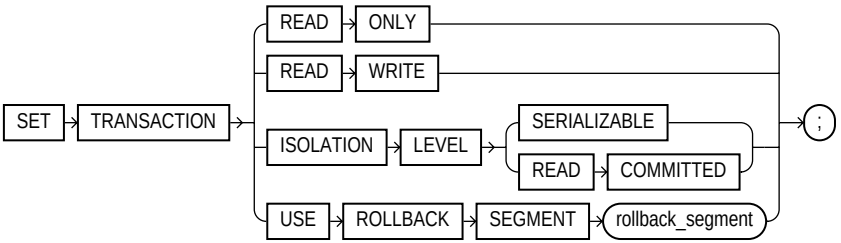
```
SET ROLE ALL EXCEPT banker;
```

To disable all roles granted to you for the current session, issue the following statement:

```
SET ROLE NONE;
```

SET TRANSACTION

Syntax



Purpose

To establish the current transaction as a read-only or read-write, establish its isolation level, or assign it to a specified rollback segment.

The operations performed by a SET TRANSACTION statement affect only your current transaction, not other users or other transactions. Your transaction ends whenever you issue a COMMIT or ROLLBACK statement. Oracle implicitly commits the current transaction before and after executing a data definition language (DDL) statement. For more information, see "COMMIT" on page 7-214 and "ROLLBACK" on page 7-537.

Prerequisites

If you use a SET TRANSACTION statement, it must be the first statement in your transaction. However, a transaction need not have a SET TRANSACTION statement.

Keywords and Parameters

READ ONLY

establishes the current transaction as a read-only transaction. This clause established **transaction-level read consistency**. For more information on this topic, see *Oracle8i Concepts*.

All subsequent queries in that transaction only see changes committed before the transaction began. Read-only transactions are useful for reports that run multiple queries against one or more tables while other users update these same tables.

Restriction: Only the following statements are permitted in a read-only transaction:

- subqueries (that is, SELECT statements without the *for_update_clause*)
- LOCK TABLE
- SET ROLE
- ALTER SESSION
- ALTER SYSTEM

READ WRITE — establishes the current transaction as a read-write transaction. This clause established **statement-level read consistency**, which is the default.

Restriction: You cannot toggle between transaction-level and statement-level read consistency in the same transaction.

ISOLATION LEVEL — specifies how transactions containing database modifications are handled.

SERIALIZABLE — specifies serializable transaction isolation mode as defined in SQL92. If a serializable transaction contains data manipulation language (DML) that attempts to update any resource that may have been updated in a transaction uncommitted at the start of the serializable transaction, then the DML statement fails.

Note: The COMPATIBLE initialization parameter must be set to 7.3.0 or higher for SERIALIZABLE mode to work.

READ COMMITTED — is the default Oracle transaction behavior. If the transaction contains DML that requires row locks held by another transaction, then the DML statement waits until the row locks are released.

USE ROLLBACK SEGMENT — assigns the current transaction to the specified rollback segment. This clause also implicitly establishes the transaction as a read-write transaction.

This clause lets you to assign transactions of different types to rollback segments of different sizes. For example:

- If no long-running queries are concurrently reading the same tables, you can assign small transactions to small rollback segments, which are more likely to remain in memory.
- You can assign transactions that modify tables that are concurrently being read by long-running queries to large rollback segments, so that the rollback information needed for the read-consistent queries is not overwritten.
- You can assign transactions that insert, update, or delete large amounts of data to rollback segments large enough to hold the rollback information for the transaction.

You cannot use the READ ONLY clause and the USE ROLLBACK SEGMENT clause in a single SET TRANSACTION statement or in different statements in the same transaction. Read-only transactions do not generate rollback information and therefore are not assigned rollback segments.

Examples

The following statements could be run at midnight of the last day of every month to count how many ships and containers the company owns. This report would not be affected by any other user who might be adding or removing ships and/or containers.

```
COMMIT;
SET TRANSACTION READ ONLY;
SELECT COUNT(*) FROM ship;
SELECT COUNT(*) FROM container;
COMMIT;
```

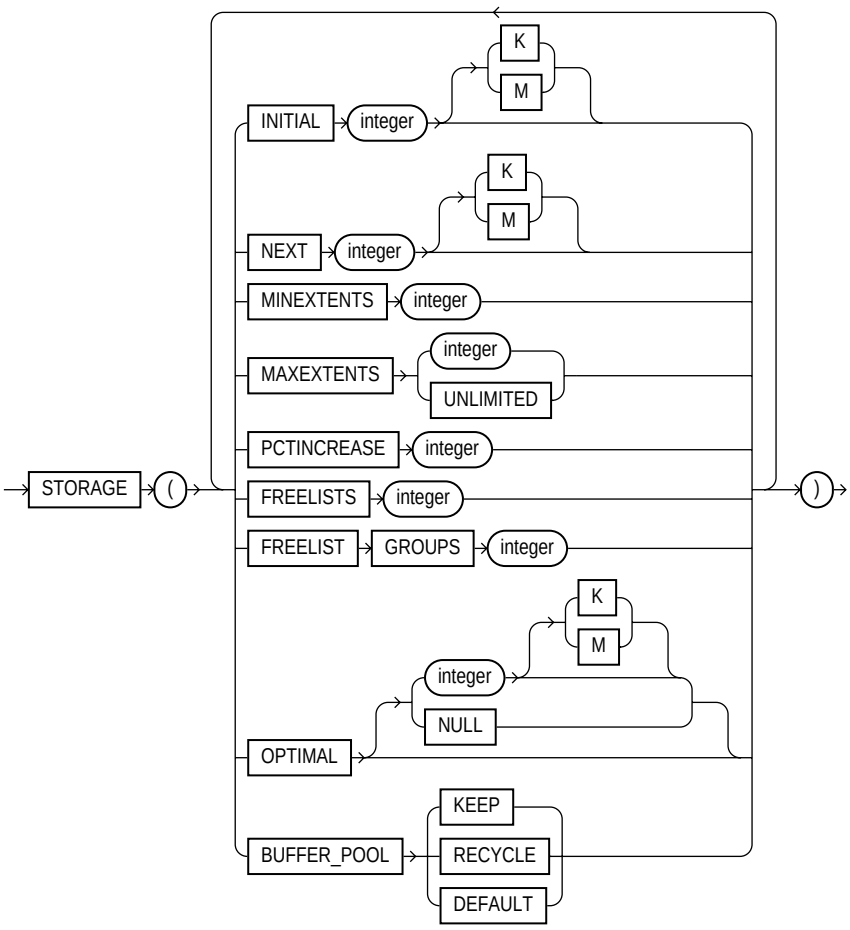
The last COMMIT statement does not actually make permanent any changes to the database. It simply ends the read-only transaction.

The following statement assigns your current transaction to the rollback segment OLTP_5:

```
SET TRANSACTION USE ROLLBACK SEGMENT oltp_5;
```

storage_clause

Syntax



Purpose

To specify storage characteristics for any of the following schema objects:

- clusters

- indexes
- rollback segments
- materialized views / snapshots
- materialized view logs / snapshot logs
- tables
- tablespaces
- partitions

Storage parameters affect both how long it takes to access data stored in the database and how efficiently space in the database is used. For a discussion of the effects of these parameters, see *Oracle8i Tuning*.

When you create a tablespace, you can specify values for the storage parameters. These values serve as default values for segments allocated in the tablespace.

When you alter a tablespace, you can change the values of storage parameters. The new values serve as default values only for subsequently allocated segments (or subsequently created objects).

Note: The *storage_clause* is interpreted differently for locally managed tablespaces. At creation, Oracle ignores MAXEXTENTS and uses the remaining parameter values to calculate the initial size of the segment. For more information, see "CREATE TABLESPACE" on page 7-394.

When you create a cluster, index, rollback segment, snapshot, snapshot log, table, or partition, you can specify values for the storage parameters for the segments allocated to these objects. If you omit any storage parameter, Oracle uses the value of that parameter specified for the tablespace.

When you alter a cluster, index, rollback segment, snapshot, snapshot log, table, or partition, you can change the values of storage parameters. The new values affect only future extent allocations.

Prerequisites

To change the value of a STORAGE parameter, you must have the privileges necessary to use the appropriate CREATE or ALTER statement.

Keywords and Parameters

INITIAL	specifies in bytes the size of the object's first extent. Oracle allocates space for this extent when you create the schema object. Use K or M to specify this size in kilobytes or megabytes. The default value is the size of 5 data blocks. The minimum value is the size of 2 data blocks for nonbitmapped segments or 3 data blocks for bitmapped segments, plus one data block for each free list group you specify (see "FREELIST GROUPS" on page 7-578). The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size for values less than 5 data blocks, and rounds up to the next multiple of 5 data blocks for values greater than 5 data blocks. **Restriction:** You cannot specify INITIAL in an ALTER statement.
NEXT	specifies in bytes the size of the next extent to be allocated to the object. Use K or M to specify the size in kilobytes or megabytes. The default value is the size of 5 data blocks. The minimum value is the size of 1 data block. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size for values less than 5 data blocks. For values greater than 5 data blocks, Oracle rounds up to a value that minimizes fragmentation, as described in *Oracle8i Concepts*. If you change the value of the NEXT parameter (that is, if you specify it in an ALTER statement), the next allocated extent will have the specified size, regardless of the size of the most recently allocated extent and the value of the PCTINCREASE parameter.
PCTINCREASE	specifies the percent by which the third and subsequent extents grow over the preceding extent. The default value is 50, meaning that each subsequent extent is 50% larger than the preceding extent. The minimum value is 0, meaning all extents after the first are the same size. The maximum value depends on your operating system. Oracle rounds the calculated size of each new extent to the nearest multiple of the data block size. If you change the value of the PCTINCREASE parameter (that is, if you specify it in an ALTER statement), Oracle calculates the size of the next extent using this new value and the size of the most recently allocated extent. **Suggestion:** If you wish to keep all extents the same size, you can prevent SMON from coalescing extents by setting the value of PCTINCREASE to 0. In general, Oracle Corporation recommends a setting of 0 as a way to minimize fragmentation and avoid the possibility of very large temporary segments during processing. **Restriction:** You cannot specify PCTINCREASE for rollback segments. Rollback segments always have a PCTINCREASE value of 0.
MINEXTENTS	specifies the total number of extents to allocate when the object is created. This parameter enables you to allocate a large amount of space when you create an object, even if the space available is not contiguous. The default and minimum value is 1, meaning that Oracle allocates only the initial extent, except for rollback segments, for which the default and minimum value is 2. The maximum value depends on your operating system.

If the MINEXTENTS value is greater than 1, then Oracle calculates the size of subsequent extents based on the values of the INITIAL, NEXT, and PCTINCREASE parameters.

Restriction: You cannot specify MINEXTENTS in an ALTER statement.

MAXEXTENTS specifies the total number of extents, including the first, that Oracle can allocate for the object. The minimum value is 1 (except for rollback segments, which always have a minimum value of 2). The default value depends on your data block size.

UNLIMITED specifies that extents should be allocated automatically as needed. Oracle Corporation recommends this setting as a way to minimize fragmentation.

However, do not use this clause for rollback segments. Rogue transactions containing inserts, updates, or deletes, that continue for a long time will continue to create new extents until a disk is full.

Caution: A rollback segment that you create without specifying the *storage_clause* has the same storage parameters as the tablespace that the rollback segment is created in. Thus, if you create the tablespace with MAXEXTENTS UNLIMITED, then the rollback segment will also have the same default.

FREELIST GROUPS for schema objects other than tablespace, specifies the number of groups of free lists for a table, partition, cluster, or index. The default and minimum value for this parameter is 1. **Use this parameter only if you are using Oracle with the Parallel Server option in parallel mode.**

Oracle uses one data block for each free list group. If you do not specify a large enough value for INITIAL to cover the minimum value plus one data block for each free list group, Oracle increases the value of INITIAL the necessary amount.

FREELISTS for objects other than tablespace, specifies the number of free lists for each of the free list groups for the table, partition, cluster, or index. The default and minimum value for this parameter is 1, meaning that each free list group contains one free list. The maximum value of this parameter depends on the data block size. If you specify a FREELISTS value that is too large, Oracle returns an error indicating the maximum value.

Restriction: You can specify the FREELISTS and the FREELIST GROUPS parameters only in CREATE TABLE, CREATE CLUSTER, and CREATE INDEX statements.

OPTIMAL is relevant only to rollback segments. It specifies an optimal size in bytes for a rollback segment. Use K or M to specify this size in kilobytes or megabytes. Oracle tries to maintain this size for the rollback segment by dynamically deallocating extents when their data is no longer needed for active transactions. Oracle deallocates as many extents as possible without reducing the total size of the rollback segment below the OPTIMAL value.

NULL specifies no optimal size for the rollback segment, meaning that Oracle never deallocates the rollback segment's extents. This is the default behavior.

	The value of OPTIMAL cannot be less than the space initially allocated for the rollback segment specified by the MINEXTENTS, INITIAL, NEXT, and PCTINCREASE parameters. The maximum value depends on your operating system. Oracle rounds values up to the next multiple of the data block size.
BUFFER_POOL	defines a default buffer pool (cache) for a schema object. All blocks for the object are stored in the specified cache. If a buffer pool is defined for a partitioned table or index, then the partitions inherit the buffer pool from the table or index definition, unless overridden by a partition-level definition.
	Note: BUFFER_POOL is not a valid clause for creating or altering tablespaces or rollback segments. For more information about using multiple buffer pools, see *Oracle8i Tuning*.
	KEEP — retains the schema object in memory to avoid I/O operations.
	RECYCLE — eliminates blocks from memory as soon as they are no longer needed, thus preventing an object from taking up unnecessary cache space.
	DEFAULT — always exists for objects not assigned to KEEP or RECYCLE.

Examples

The following statement creates a table and provides storage parameter values:

```
CREATE TABLE dept
    (deptno     NUMBER(2),
     dname      VARCHAR2(14),
     loc        VARCHAR2(13) )
     STORAGE  ( INITIAL 100K  NEXT     50K
                MINEXTENTS 1  MAXEXTENTS 50 );
```

Oracle allocates space for the table based on the STORAGE parameter values as follows:

- The MINEXTENTS value is 1, so Oracle allocates 1 extent for the table upon creation.
- The INITIAL value is 100K, so the first extent's size is 100 kilobytes.
- If the table data grows to exceed the first extent, Oracle allocates a second extent. The NEXT value is 50K, so the second extent's size would be 50 kilobytes.
- If the table data subsequently grows to exceed the first two extents, Oracle allocates a third extent. The PCTINCREASE value is 5, so the calculated size of the third extent is 5% larger than the second extent, or 52.5 kilobytes. If the data block size is 2 kilobytes, Oracle rounds this value to 52 kilobytes.

If the table data continues to grow, Oracle allocates more extents, each 5% larger than the previous one.

- The MAXEXTENTS value is 50, so Oracle can allocate as many as 50 extents for the table.

The following statement creates a rollback segment and provides storage parameter values:

```
CREATE ROLLBACK SEGMENT rsone
    STORAGE ( INITIAL  10K  NEXT 10K
              MINEXTENTS 2  MAXEXTENTS 25
               OPTIMAL 50K );
```

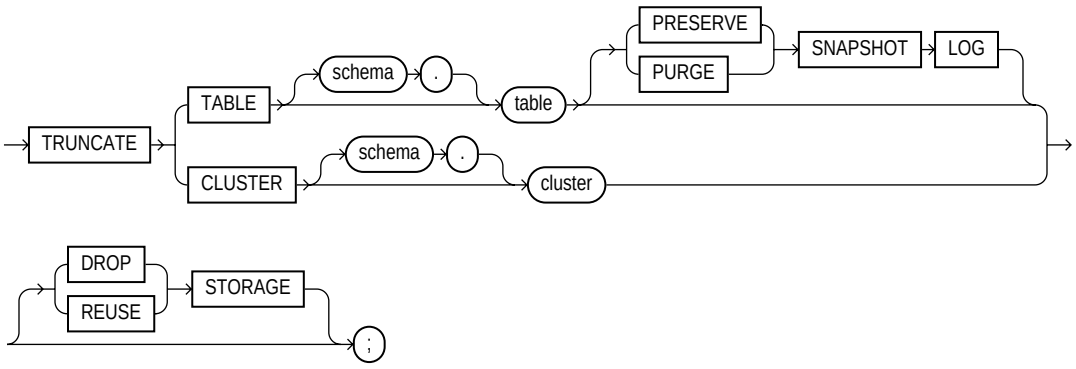
Oracle allocates space for the rollback segment based on the STORAGE parameter values as follows:

- The MINEXTENTS value is 2, so Oracle allocates 2 extents for the rollback segment upon creation.
- The INITIAL value is 10K, so the first extent's size is 10 kilobytes.
- The NEXT value is 10K, so the second extent's size is 10 kilobytes.
- If the rollback data exceeds the first two extents, Oracle allocates a third extent. The PCTINCREASE value for rollback segments is always 0, so the third and subsequent extents are the same size as the second extent, 10 kilobytes.
- The MAXEXTENTS value is 25, so Oracle can allocate as many as 25 extents for the rollback segment.
- The OPTIMAL value is 50K, so Oracle deallocates extents if the rollback segment exceeds 50 kilobytes. Oracle deallocates only extents that contain data for transactions that are no longer active.

TRUNCATE

WARNING: You cannot roll back a TRUNCATE statement.

Syntax



Purpose

To remove all rows from a table or cluster and reset the STORAGE parameters to the values when the table or cluster was created.

Deleting rows with the TRUNCATE statement can be more efficient than dropping and re-creating a table. Dropping and re-creating a table invalidates the table's dependent objects, requires you to regrant object privileges on the table, and requires you to re-create the table's indexes, integrity constraint, and triggers and respecify its storage parameters. Truncating has none of these effects.

See also "DELETE" on page 7-438, "DROP CLUSTER" on page 7-446, and "DROP TABLE" on page 7-475.

Prerequisites

The table or cluster must be in your schema or you must have DROP ANY TABLE system privilege.

Keywords and Parameters

TABLE

specifies the schema and name of the table to be truncated. This table cannot be part of a cluster. If you omit *schema*, Oracle assumes the table is in your own cluster.

You can truncate index-organized tables and temporary tables. When you truncate a temporary table, only the rows created during the current session are truncated.

The table's storage parameter NEXT is changed to be the size of the last extent deleted from the segment in the process of truncation.

Oracle also automatically truncates and resets any existing UNUSABLE indicators for the following indexes on *table*: range and hash partitions of local indexes and subpartitions of local indexes.

If *table* is not empty, Oracle marks UNUSABLE all nonpartitioned indexes and all partitions of global partitioned indexes on the table.

For a domain index, this statement invokes the appropriate truncate routine to truncate the domain index data. For more information, see *Oracle8i Data Cartridge Developer's Guide*.

If *table* (whether it is a regular or index-organized table) contains LOB columns, all LOB data and LOB index segments will be truncated.

If *table* is partitioned, all partitions or subpartitions, as well as the LOB data and LOB index segments for each partition or subpartition, will be truncated.

Note: When you truncate a table, Oracle automatically deletes all data in the table's indexes and any materialized view direct-load INSERT information held in association with the table. (This information is independent of any materialized view / snapshot log.) If this direct-load INSERT information is deleted, an incremental refresh of the materialized view may lose data.

Restrictions:

- You cannot individually truncate a table that is part of a cluster. You must either truncate the cluster, delete all rows from the table, or drop and re-create the table.
- You cannot truncate the parent table of an enabled referential integrity constraint. You must disable the constraint before truncating the table. (An exception is that you may truncate the table if the integrity constraint is self-referential.)
- You cannot truncate a table if any domain indexes defined on any of its columns are marked LOADING or FAILED.

SNAPSHOT LOG

specifies whether a snapshot log defined on the table is to be preserved or purged when the table is truncated. This clause allows snapshot master tables to be reorganized through export / import without affecting the ability of primary-key snapshots defined on the master to be fast refreshed. To support continued fast refresh of primary-key snapshots, the snapshot log must record primary-key information. For more information about snapshot logs and the TRUNCATE statement, see *Oracle8i Replication*.

	PRESERVE	specifies that any snapshot log should be preserved when the master table is truncated. This is the default.
	PURGE	specifies that any snapshot log should be purged when the master table is truncated.
CLUSTER	specifies the schema and name of the cluster to be truncated. You can truncate only an indexed cluster, not a hash cluster. If you omit *schema*, Oracle assumes the table is in your own cluster. When you truncate a cluster, Oracle also automatically deletes all data in the cluster's tables' indexes.	
DROP STORAGE	deallocates all space from the deleted rows from the table or cluster except the space allocated by the table's or cluster's MINEXTENTS parameter. This space can subsequently be used by other objects in the tablespace. This is the default.	
REUSE STORAGE	retains the space from the deleted rows allocated to the table or cluster. Storage values are not reset to the values when the table or cluster was created. This space can subsequently be used only by new data in the table or cluster resulting from inserts or updates. The DROP STORAGE clause and REUSE STORAGE clause also apply to the space freed by the data deleted from associated indexes.	
	Note: If you have specified more than one free list for the object you are truncating, the REUSE STORAGE clause also removes any mapping of free lists to instances, and resets the high-water mark to the beginning of the first extent.	

Examples

The following statement deletes all rows from the EMP table and returns the freed space to the tablespace containing EMP:

```
TRUNCATE TABLE emp;
```

The above statement also deletes all data from all indexes on EMP and returns the freed space to the tablespaces containing them.

The following statement deletes all rows from all tables in the CUST cluster, but leaves the freed space allocated to the tables:

```
TRUNCATE CLUSTER cust REUSE STORAGE
```

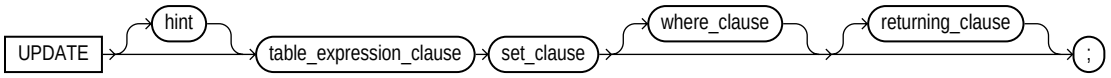
The above statement also deletes all data from all indexes on the tables in CUST.

The following statements are examples of truncate statements that preserve snapshot logs:

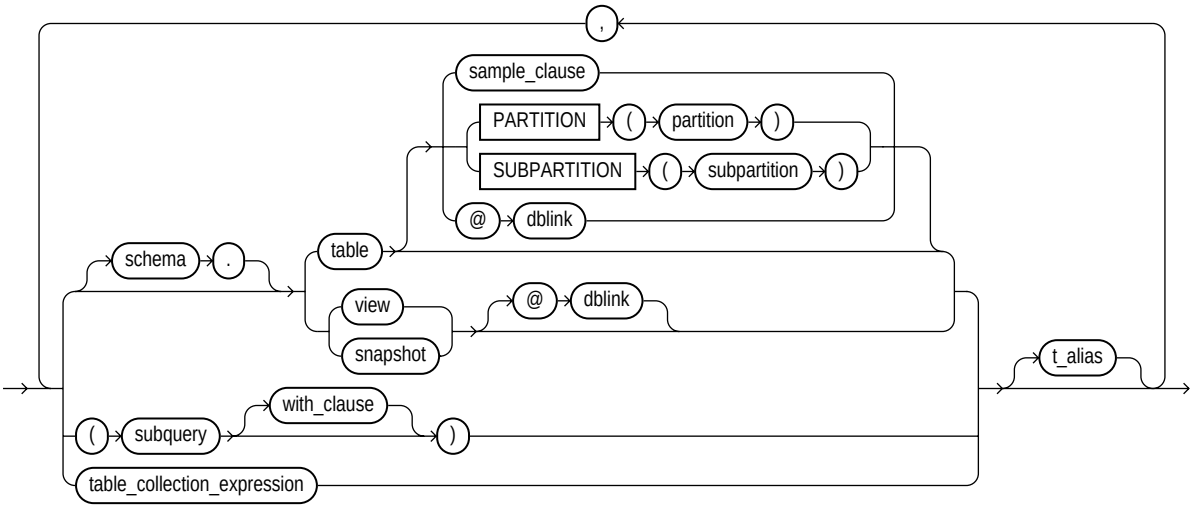
```
TRUNCATE TABLE emp PRESERVE SNAPSHOT LOG;
TRUNCATE TABLE stock;
```

UPDATE

Syntax

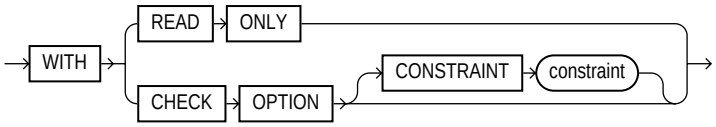


table_expression_clause::=



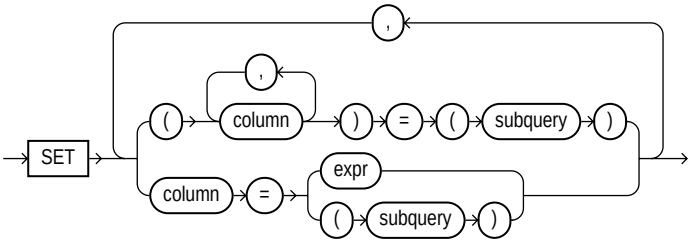
subquery: see "SELECT and Subqueries" on page 7-541.

with_clause::=

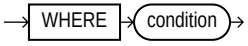


table_collection_expression::=

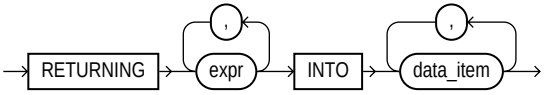
set_clause::=



where_clause::=



returning_clause::=



Purpose

To change existing values in a table or in a view's base table.

Prerequisites

For you to update values in a table, the table must be in your own schema or you must have UPDATE privilege on the table.

For you to update values in the base table of a view,

- You must have UPDATE privilege on the view, and
- Whoever owns the schema containing the view must have UPDATE privilege on the base table.

If the SQL92_SECURITY initialization parameter is set to TRUE, then you must have SELECT privilege on the table whose column values you are referencing (such as the columns in a *where_clause*) to perform an UPDATE.

The UPDATE ANY TABLE system privilege also allows you to update values in any table or any view's base table.

Keywords and Parameters

hint is a comment that passes instructions to the optimizer on choosing an execution plan for the statement. For the syntax and description of hints, see "Hints" on page 2-58 and *Oracle8i Tuning*.

You can place a parallel hint immediately after the UPDATE keyword to parallelize both the underlying scan and UPDATE operations. For detailed information about parallel DML, see *Oracle8i Tuning*, *Oracle8i Parallel Server Concepts and Administration*, and *Oracle8i Concepts*.

table_expression_clause

schema is the schema containing the table or view. If you omit *schema*, Oracle assumes the table or view is in your own schema.

table | *view* | *subquery* is the name of the table or view, or the columns returned by a subquery, to be updated. Issuing an UPDATE statement against a table fires any UPDATE triggers associated with the table. If you specify *view*, Oracle updates the view's base table.

If *table* (or the base table of *view*) contains one or more domain index columns, this statement executes the appropriate indextype update routine. For more information on these routines, see *Oracle8i Data Cartridge Developer's Guide*.

Restrictions:

- You cannot execute this statement if *table* (or the base table of *view*) contains any domain indexes marked LOADING or FAILED.
- You cannot specify the *sample_clause* in an UPDATE statement.
- You cannot specify the *order_by_clause* in the subquery of the *table_expression_clause*.

- You cannot update a view except with INSTEAD OF triggers if the view's defining query contains one of the following constructs:

  - A set operator

  - A DISTINCT operator

  - An aggregate function

  - A GROUP BY, ORDER BY, CONNECT BY, or START WITH clause

  - A collection expression in a SELECT list

  - A subquery in a SELECT list

  - Joins (with some exceptions). See *Oracle8i Administrator's Guide* for details.

- If a view was created with the WITH CHECK OPTION, you can update the view only if the resulting data satisfies the view's defining query.

- If you specify an index, index partition, or index subpartition that has been marked UNUSABLE, the UPDATE statement will fail unless the SKIP_UNUSABLE_INDEXES parameter has been set to TRUE. For more information, see "ALTER SESSION" on page 7-78

PARTITION *(partition)* | SUBPARTITION *(subpartition)*

specifies the name of the partition or subpartition within *table* targeted for updates. You need not specify the partition name when updating values in a partitioned table. However in some cases specifying the partition name can be more efficient than a complicated *where_clause*.

dblink is a complete or partial name of a database link to a remote database where the table or view is located. For information on referring to database links, see "Referring to Objects in Remote Databases" on page 2-74. You can use a database link to update a remote table or view only if you are using Oracle's distributed functionality.

If you omit *dblink*, Oracle assumes the table or view is on the local database.

with_clause restricts the subquery in one of the following ways:

- WITH READ ONLY specifies that the subquery cannot be updated.

- WITH CHECK OPTION specifies that Oracle prohibits any changes to that table that would produce rows that are not included in the subquery. See the WITH CHECK OPTION Example on page 7-558.

table_collection_expression informs Oracle that the collection value expression should be treated as a table. You can use a *table_collection_expression* to update rows in one table based on rows from another table. For example, you could roll up four quarterly sales tables into a yearly sales table.

collection_expression is a subquery that selects a nested table column from *table* or *view*.

Note: In earlier releases of Oracle, *table_collection_expr* was expressed as "THE subquery". That usage is now deprecated.

t_alias provides a **correlation name** for the table, view, or subquery to be referenced elsewhere in the statement.

Note: This alias is **required** if the *table_expression_clause* references any object type attributes or object type methods.

set_clause

column is the name of a column of the table or view that is to be updated. If you omit a column of the table from the *set_clause*, that column's value remains unchanged.

Restrictions:

- If *column* refers to a LOB object attribute, you cannot update it with a literal. Also, before you can update a LOB value, you must lock the row containing the LOB. See the LOB Locking Example on page 7-556.
- If *column* is part of the partitioning key of a partitioned table, UPDATE will fail if you change a value in the column that would move the row to a different partition or subpartition, unless you enable row movement. See the *row_movement_clause* of "CREATE TABLE" on page 7-359 or "ALTER TABLE" on page 7-113.

subquery is a subquery that returns exactly one row for each row updated.

- If you specify only one column in the *set_clause*, the subquery can return only one value.
- If you specify multiple columns in the *set_clause*, the subquery must return as many values as you have specified columns.

If the subquery returns no rows, then the column is assigned a null. See also "SELECT and Subqueries" on page 7-541 and "Using Subqueries" on page 5-23.

expr is the new value assigned to the corresponding column. This expression can contain host variables and optional indicator variables. See the syntax description in "Expressions" on page 5-1.

Note: If you insert string literals into a RAW column, during subsequent queries, Oracle will perform a full table scan rather than using any index that might exist on the RAW column.

where_clause restricts the rows updated to those for which the specified *condition* is TRUE. If you omit this clause, Oracle updates all rows in the table or view. See the syntax description of "Conditions" on page 5-13.

The *where_clause* determines the rows in which values are updated. If you do not specify the *where_clause*, all rows are updated. For each row that satisfies the *where_clause*, the columns to the left of the equals (=) operator in the *set_clause* are set to the values of the corresponding expressions on the right. The expressions are evaluated as the row is updated.

returning_clause retrieves the rows affected by the UPDATE statement.

- When you are updating a single row, this clause can retrieve column expressions that use the updated columns of the row, rowid, and REFs to the updated row and store them in PL/SQL variables or bind variables.
- When you are updating multiple rows, this clause can stored the values from expressions, rowid, and REFs involving the updated rows in bind arrays.
- You can also use UPDATE with a *returning_clause* to update from views with single base tables.

expr list	is some of the syntax descriptions in "Expressions" on page 5-1. You must specify a column expression in the *expr* list for each variable in the *data_item* list.
INTO	indicates that the values of the changed rows are to be stored in the *data_item* variable(s) specified in *data_item* list.
data_item	is a PL/SQL variable or bind variable which stores the retrieved *expr* value in the *expr* list.

Restrictions:

- You cannot use this clause with parallel DML or with remote objects.
- You cannot retrieve LONG types with this clause.

Examples

Simple Examples The following statement gives null commissions to all employees with the job TRAINEE:

```
UPDATE emp
    SET comm = NULL
    WHERE job = 'TRAINEE';
```

The following statement promotes JONES to manager of Department 20 with a $1,000 raise (assuming there is only one JONES):

```
UPDATE emp
    SET job = 'MANAGER', sal = sal + 1000, deptno = 20
    WHERE ename = 'JONES';
```

The following statement increases the balance of bank account number 5001 in the ACCOUNTS table on a remote database accessible through the database link BOSTON:

```
UPDATE accounts@boston
    SET balance = balance + 500
    WHERE acc_no = 5001;
```

PARTITION Example The following example updates values in a single partition of the SALES table:

```
UPDATE sales PARTITION (feb96) s
   SET s.account_name = UPPER(s.account_name);
```

Complex Example This example shows the following syntactic constructs of the UPDATE statement:

- Both forms of the *set_clause* together in a single statement
- A correlated subquery
- A *where_clause* to limit the updated rows

```
UPDATE emp a
    SET deptno =
        (SELECT deptno
            FROM dept
            WHERE loc = 'BOSTON'),
        (sal, comm) =
        (SELECT 1.1*AVG(sal), 1.5*AVG(comm)
          FROM emp b
          WHERE a.deptno = b.deptno)
    WHERE deptno IN
        (SELECT deptno
          FROM dept
          WHERE loc = 'DALLAS'
              OR loc = 'DETROIT');
```

The above UPDATE statement performs the following operations:

- Updates only those employees who work in Dallas or Detroit
- Sets DEPTNO for these employees to the DEPTNO of Boston
- Sets each employee's salary to 1.1 times the average salary of their department
- Sets each employee's commission to 1.5 times the average commission of their department

Correlated Update Example The following example updates particular rows of the PROJS nested table corresponding to the department whose department equals 123:

```
UPDATE TABLE(SELECT projs
           FROM dept d WHERE d.dno = 123)  p
  SET p.budgets = p.budgets + 1
```

```
  WHERE p.pno IN (123, 456);
```

RETURNING Example The following example returns values from the updated row and stores the result in PL/SQL variables BND1, BND2, BND3:

```
UPDATE emp
  SET job ='MANAGER', sal = sal + 1000, deptno = 20
  WHERE ename = 'JONES'
  RETURNING sal*0.25, ename, deptno INTO bnd1, bnd2, bnd3;
```

A

Syntax Diagrams

One picture is worth a thousand words.

Anonymous

Syntax diagrams are drawings that illustrate valid SQL syntax. To read a diagram, trace it from left to right, in the direction shown by the arrows.

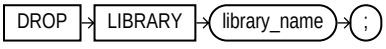
Commands and other keywords appear in UPPERCASE inside rectangles. Type them exactly as shown in the rectangles. Parameters appear in lowercase inside ovals. Variables are used for the parameters. Punctuation, operators, delimiters, and terminators appear inside circles.

If the syntax diagram has more than one path, you can choose any path to travel.

If you have the choice of more than one keyword, operator, or parameter, your options appear in a vertical list.

Required Keywords and Parameters

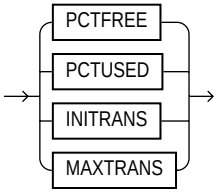
Required keywords and parameters can appear singly or in a vertical list of alternatives. Single required keywords and parameters appear on the *main path*, that is, on the horizontal line you are currently traveling. In the following example, *library_name* is a required parameter:



If there is a library named *HQ_LIB*, then, according to the diagram, the following statement is valid:

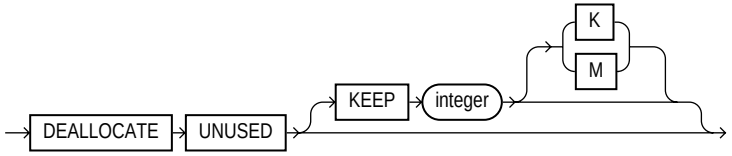
```
DROP LIBRARY hq_lib;
```

If multiple keywords or parameters appear in a vertical list that intersects the main path, one of them is required. That is, you must choose one of the keywords or parameters, but not necessarily the one that appears on the main path. In the following example, you must choose one of the four settings:



Optional Keywords and Parameters

If keywords and parameters appear in a vertical list *above* the main path, they are optional. In the following example, instead of traveling down a vertical line, you can continue along the main path:

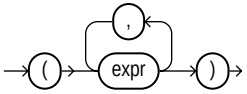


According to the diagram, all of the following statements are valid:

```
DEALLOCATE UNUSED;
DEALLOCATE UNUSED KEEP 1000;
DEALLOCATE UNUSED KEEP 10M;
```

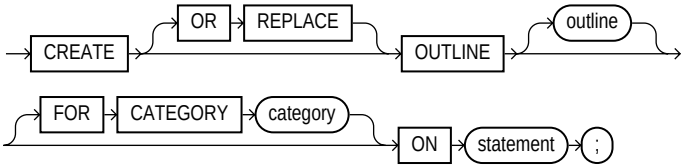
Syntax Loops

Loops let you repeat the syntax within them as many times as you like. In the following example, after choosing one expression, you can go back repeatedly to choose another, separated by commas.



Multipart Diagrams

Read a multipart diagram as if all the main paths were joined end to end. The following example is a two-part diagram:



According to the diagram, the following statement is valid:

```
CREATE OUTLINE ON UPDATE;
```

Database Objects

The names of Oracle identifiers, such as tables and columns, must not exceed 30 characters in length. The first character must be a letter, but the rest can be any combination of letters, numerals, dollar signs ($), pound signs (#), and underscores (_).

However, if an Oracle identifier is enclosed by double quotation marks ("), it can contain any combination of legal characters, including spaces but excluding quotation marks.

Oracle identifiers are not case-sensitive except when enclosed by double quotation marks.

For more information, see "Schema Object Naming Rules" on page 2-67.

B

Oracle and Standard SQL

High thoughts must have high language.

Aristophanes, *Frogs*

This appendix discusses Oracle's conformance to the SQL standards established by industry standards governing bodies. It also described how to locate extensions to standard SQL with the FIPS Flagger.

Conformance with Standard SQL

This section declares Oracle's conformance to the SQL standards established by these organizations:

- American National Standards Institute (ANSI)
- International Standards Organization (ISO)
- United States Federal Government

ANSI and ISO Compliance

Oracle8*i* complies at the Entry level as defined in the ANSI document, X3.135-1992, "Database Language SQL." You can obtain a copy of the ANSI standard from this address:

American National Standards Institute
1430 Broadway
New York, NY 10018 USA

The ANSI and ISO SQL standards require conformance claims to state the type of conformance and the implemented facilities. The Oracle server, Oracle Precompilers for C/C++ Release 8.1, Oracle Precompiler for Cobol Release 8.1, and

SQL*Module for ADA Release 8.0.4 provide conformance with the ANSI X3.135-1992 / ISO 9075-1992 standard:

- Compliance at Entry Level (including both SQL-DDL and SQL-DML)
- Module Language for ADA
- Embedded SQL C
- Embedded SQL COBOL

In addition to full compliance at the Entry level, Oracle complies partially at the Transitional, Intermediate, and Full levels as described in Table B–1 (including both SQL-DDL and SQL-DML).

Table B–1 Oracle Compliance at Transitional, Intermediate, and Full Levels

Level	SQL92 Feature (number and name)
Transitional	7. TRIM function
	8. UNION in views
	9. Implicit numeric casting
	10. Implicit character casting
	13. Grouped operations
	14. Qualified * in SELECT list
	15. Lowercase identifiers
	16. PRIMARY KEY enhancement
	18. Multiple module support
	21. INSERT expressions
Intermediate	31. Schema definition statement
	42. National character
	48. Expanded null predicate
Full	60. Trailing underscore
	62. Referential name order

FIPS Compliance

Oracle complies completely with FIPS PUB 127-2 for Entry SQL. In addition, the following information is provided for Section 16, “Special Procurement Considerations.”

Section 16.2 Programming Language Interfaces

The Oracle precompilers support the use of embedded SQL in C and COBOL. SQL*Module supports the use of Module Language in ADA.

Section 16.3 Style of Language Interface

Oracle with SQL*Module supports Module Language for Ada. Oracle with the Oracle precompilers supports C and COBOL. The specific languages supported depend on your operating system.

Section 16.5 Interactive Direct SQL

Oracle8*i* with SQL*Plus Version 3.1 (as well as other Oracle tools) supports "direct invocation" of the following SQL statements, meeting the requirements of FIPS PUB 127-2:

- CREATE TABLE statement
- CREATE VIEW statement
- GRANT statement
- INSERT statement
- SELECT statement, with ORDER BY clause but not INTO clause
- UPDATE statement: searched
- DELETE statement: searched
- COMMIT WORK statement
- ROLLBACK WORK statement

Most other SQL statements described in this reference are also supported interactively.

Section 16.6 Sizing for Database Constructs

Table B–2 lists requirements identified in FIPS PUB 127-1 and how they are met by Oracle8*i*.

Table B–2 Sizing for Database Constructs

Database Constructs	FIPS	Oracle8*i*
Length of an identifier (in bytes)	18	30
Length of CHARACTER datatype (in bytes)	240	2000
Decimal precision of NUMERIC datatype	15	38
Decimal precision of DECIMAL datatype	15	38
Decimal precision of INTEGER datatype	9	38
Decimal precision of SMALLINT datatype	4	38
Binary precision of FLOAT datatype	20	126
Binary precision of REAL datatype	20	63
Binary precision of DOUBLE PRECISION datatype	30	126
Columns in a table	100	1000
Values in an INSERT statement	100	1000
SET clauses in an UPDATE statement(a)	20	1000
Length of a row(b,c)	2,000	2,000,000
Columns in a UNIQUE constraint	6	32
Length of a UNIQUE constraint(b)	120	(d)
Length of foreign key column list(b)	120	(d)
Columns in a GROUP BY clause	6	255(e)
Length of GROUP BY column list	120	(e)
Sort specifications in ORDER BY clause	6	255(e)
Length of ORDER BY column list	120	(e)
Columns in a referential integrity constraint	6	32
Tables referenced in a SQL statement	15	No limit
Cursors simultaneously open	10	(f)
Items in a SELECT list	100	1000

Table B–2 (Cont.) Sizing for Database Constructs

(a) The number of SET clauses in an UPDATE statement refers to the number items separated by commas following the SET keyword.

(b) The FIPS PUB defines the length of a collection of columns to be the sum of: twice the number of columns, the length of each character column in bytes, decimal precision plus 1 of each exact numeric column, binary precision divided by 4 plus 1 of each approximate numeric column.

(c) The Oracle limit for the maximum row length is based on the maximum length of a row containing a LONG value of length 2 gigabytes and 999 VARCHAR2 values, each of length 4000 bytes: 2(254) + 231 + (999(4000)).

(d) The Oracle limit for a UNIQUE key is half the size of an Oracle data block (specified by the initialization parameter DB_BLOCK_SIZE) minus some overhead.

(e) Oracle places no limit on the number of columns in a GROUP BY clause or the number of sort specifications in an ORDER BY clause. However, the sum of the sizes of all the expressions in either a GROUP BY clause or an ORDER BY clause is limited to the size of an Oracle data block (specified by the initialization parameter DB_BLOCK_SIZE) minus some overhead.

(f) The Oracle limit for the number of cursors simultaneously opened is specified by the initialization parameter OPEN_CURSORS. The maximum value of this parameter depends on the memory available on your operating system and exceeds 100 in all cases.

Section 16.7 Character Set Support

Oracle supports the ASCII character set (FIPS PUB 1-2) on most computers and the EBCDIC character set on IBM mainframe computers. Oracle supports both single-byte and multibyte character sets.

Oracle Extensions to Standard SQL

Oracle supports numerous features that extend beyond standard SQL. In your Oracle applications, you can use these extensions just as you can use Entry SQL92.

If you are concerned with the portability of your applications to other implementations of SQL, use Oracle's FIPS Flagger to locate Oracle extensions to Entry SQL92 in your embedded SQL programs. The FIPS Flagger is part of the Oracle precompilers and the SQL*Module compiler. For information on how to use the FIPS Flagger, see *Pro*COBOL Precompiler Programmer's Guide* and *Pro*C/C++ Precompiler Programmer's Guide.*

C

Oracle Reserved Words

The words I use are everyday words and yet are not the same!

Paul Claudel, *La Muse Qui Est la Grace*

This appendix lists Oracle reserved words. Words followed by an asterisk (*) are also ANSI reserved words.

Note: In addition to the following reserved words, Oracle uses system-generated names beginning with "SYS_" for implicitly generated schema objects and subobjects. Oracle discourages you from using this prefix in the names you explicitly provide to your schema objects and subobjects to avoid possible conflict in name resolution.

Table C–1 Oracle Reserved Words

ACCESS	CHAR	DEFAULT
ADD	CHECK	DELETE
ALL	CLUSTER	DESC
ALTER	COLUMN	DISTINCT
AND	COMMENT	DROP
ANY	COMPRESS	ELSE
AS	CONNECT	EXCLUSIVE
ASC	CREATE	EXISTS
AUDIT	CURRENT	FILE
BETWEEN	DATE	FLOAT
BY	DECIMAL	FOR

Table C–1 Oracle Reserved Words

FROM	NOT	SHARE
GRANT	NOWAIT	SIZE
GROUP	NULL	SMALLINT
HAVING	NUMBER	START
IDENTIFIED	OF	SUCCESSFUL
IMMEDIATE	OFFLINE	SYNONYM
IN	ON	SYSDATE
INCREMENT	ONLINE	TABLE
INDEX	OPTION	THEN
INITIAL	OR	TO
INSERT	ORDER	TRIGGER
INTEGER	PCTFREE	UID
INTERSECT	PRIOR	UNION
INTO	PRIVILEGES	UNIQUE
IS	PUBLIC	UPDATE
LEVEL	RAW	USER
LIKE	RENAME	VALIDATE
LOCK	RESOURCE	VALUES
LONG	REVOKE	VARCHAR
MAXEXTENTS	ROW	VARCHAR2
MINUS	ROWID	VIEW
MLSLABEL	ROWNUM	WHENEVER
MODE	ROWS	WHERE
MODIFY	SELECT	WITH
NOAUDIT	SESSION	
NOCOMPRESS	SET	

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y